Java™

Полностью обновленное издание для Java SE 9 (JDK 9) и SE 10 (JDK 10)

Java

Руководство для начинающих

7-е издание

Современные методы создания, компиляции и выполнения программ на Java

Герберт Шилдт

Руководство для начинающих

Java

7-е издание

A Beginner's Guide

Java™

Seventh Edition

Herbert Schildt

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Руководство для начинающих

Java

7-е издание

Герберт Шилдт

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Ш57
УДК 681.3.07

Компьютерное издательство "Диалектика"

Перевод с английского и редакция *А.П. Сергеева*

Под редакцией *В.Р. Гинзбурга*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Шилдт, Герберт

Ш57 Java: руководство для начинающих, 7-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 816 с. : ил. — Парал. тит. англ.

ISBN 978-5-6041394-5-5 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства McGraw-Hill Education.

Authorized Russian translation of the English edition of *Java: A Beginners Guide, Seventh Edition* (ISBN 978-1-259-58931-7) © 2018 by McGraw-Hill Education

This translation is published and sold by permission of McGraw-Hill Education, which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Герберт Шилдт

Java: руководство для начинающих, 7-е издание

Подписано в печать 05.10.2018.
Формат 70x100/16. Гарнитура NewtonC.
Усл. печ. л. 65,79. Уч.-изд. л. 37,9.
Тираж 500 экз. Заказ № 9822

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-6041394-5-5 (рус.)
ISBN 978-1-259-58931-7 (англ.)

© 2019 ООО "Диалектика"
© 2018 by McGraw-Hill Education

Оглавление

Содержание

Об авторе

Герберт Шилдт — общепризнанный эксперт по языку Java, автор множества бестселлеров, посвященных программированию, за плечами которого — более чем 30 лет писательской деятельности. Его книги переведены на многие языки и продаются миллионными тиражами. Интересуется всем, что связано с компьютерами, но основная сфера его интересов — языки программирования. Окончил Университет штата Иллинойс и там же получил ученую степень. Посетите его сайт `www.HerbSchildt.com`.

О техническом редакторе

Дэнни Кауард участвовал в разработке всех версий Java. Под его руководством проходило внедрение технологии Java Servlets в первый и последующий выпуски платформы Java EE, разрабатывались веб-службы для платформы Java ME и осуществлялось стратегическое планирование платформы Java SE 7. Он один из авторов технологии JavaFX, а в последнее время занимался проектированием Java WebSocket API — одного из наиболее значительных нововведений стандарта Java EE 7. Благодаря огромному опыту написания программ на Java, участию в проектировании библиотек классов совместно с отраслевыми экспертами, а также многолетнему членству в исполнительном комитете JSP (Java Community Process) Дэнни обладает уникальным объемом знаний по технологиям Java. Он также автор двух книг по Java. Имеет докторскую степень по математике, полученную в Оксфордском университете.

Введение

Цель этой книги — научить читателей основам программирования на Java. В ней применяется пошаговый подход к освоению языка, основанный на анализе многочисленных примеров, разработке несложных проектов и закреплении полученных знаний путем ответа на вопросы и выполнения упражнений для самопроверки. Изучение Java не потребует от читателей предыдущего опыта программирования. Книга начинается с рассмотрения элементарных понятий, таких как компиляция и запуск программ. Затем обсуждаются ключевые слова, языковые средства и конструкции, составляющие основу языка Java. Далее изучаются более сложные концепции, включая многопоточное программирование, обобщения, лямбда-выражения и модули. Завершается книга знакомством с библиотеками Swing и JavaFX. Все это позволит читателям овладеть основами программирования на Java.

Впрочем, эта книга — лишь первый шаг на пути к освоению Java, поскольку для профессионального программирования на Java нужно знать не только составные элементы языка, но и многочисленные библиотеки и инструменты, существенно упрощающие процесс разработки программ. После прочтения книги вы получите достаточно знаний, чтобы приступить к изучению всех остальных аспектов Java.

Эволюция Java

Немногие языки могут похвастаться тем, что им удалось изменить всеобщее представление о программировании. Но и в этой “элитной” группе один язык выделяется среди остальных. Его влияние очень быстро почувствовали все программисты. Речь, конечно же, идет о Java. Не будет преувеличением сказать, что выпуск в 1995 году компанией Sun Microsystems версии Java 1.0 вызвал настоящую революцию в программировании. В результате Интернет стал по-настоящему интерактивной средой. Фактически Java установил новый стандарт среди языков программирования.

С годами Java все больше совершенствовался. В отличие от многих других языков, в которых новые средства внедрялись относительно медленно, Java всегда находился на переднем крае разработки языков программирования. Одной из причин, позволивших добиться этого, послужило формирование вокруг Java плодотворной атмосферы, способствовавшей внедрению новых идей. В результате язык Java постоянно развивался: одни его изменения были незначительными, а другие — весьма существенными.

Первым крупным обновлением Java стала версия 1.1. Изменения в ней были более значительными, чем это обычно подразумевает смена младшего номера в версии платформы. В Java 1.1 появилось множество библиотечных элементов, были переопределены средства обработки событий и перекомпонованы многие функциональные средства исходной библиотеки версии 1.0.

Следующим этапом развития языка стала платформа Java 2, где цифра 2 означает "второе поколение". Ее появление стало поворотным событием, ознаменовавшим начало новой эпохи. Первым выпуском Java 2 стала версия 1.2. На первый взгляд, несоответствие номеров в обозначениях Java 2 и версии 1.2 может показаться странным. Дело в том, что номером 1.2 сначала обозначались библиотеки Java и только затем — весь выпуск. Компания Sun перекомпоновала Java в J2SE (Java 2 Platform Standard Edition), и с тех пор номера версий стали относиться именно к этому продукту.

Затем появилась версия J2SE 1.3, в которую были внесены первые значительные изменения по сравнению с первоначальным выпуском Java 2. Новые функциональные средства были в основном добавлены к имеющимся, улучшив возможности среды разработки. Версия J2SE 1.4 стала очередным этапом в развитии Java. Она содержала новые важные средства, такие как цепочки исключений, канальный ввод-вывод и ключевое слово `assert`.

Следующая версия, J2SE 5, стала вторым революционным преобразованием Java. В отличие от большинства предыдущих модернизаций, которые сводились к важным, но предсказуемым усовершенствованиям, в J2SE 5 были существенно расширены рамки применения и функциональные возможности языка, а также повышена его производительность. Для более ясного представления о масштабах изменений, внесенных в версии J2SE 5, ниже перечислены ключевые новинки:

- обобщения;
- автоупаковка и автораспаковка;
- перечисления;
- усовершенствованный вариант цикла `for` (в стиле `for-each`);
- список аргументов переменной длины;
- статический импорт;
- аннотации.

Здесь не указаны второстепенные дополнения и поэтапные изменения, характерные для перехода к новой версии. Каждый элемент этого списка представляет собой значительное усовершенствование Java. Для поддержки одних нововведений, в том числе обобщений, циклов типа `for-each` и списков аргументов переменной длины, понадобилось добавить в язык новые синтаксические конструкции. Другие нововведения, такие как автоупаковка и автораспаковка, повлияли на семантику языка. Наконец, аннотации открыли совершенно новые возможности для программирования.

Особая значимость описанных новшеств проявилась в том, что новая версия получила номер “5”. Можно было ожидать, что номером очередной версии Java будет 1.5. Однако нововведения оказались настолько существенными, что переход от версии 1.4 к 1.5 не отражал бы масштабов внесенных изменений. Поэтому разработчики из компании Sun решили увеличить номер версии до 5, подчеркнув тем самым важность нововведений. В итоге новая версия получила название J2SE 5, а комплект разработчика приложений стал называться JDK 5. Но ради согласованности с предыдущими версиями было решено использовать 1.5 в качестве внутреннего номера версии, который иногда называют *номером версии разработки*. Цифра “5” в J2SE 5 означает *номер версии программного продукта*.

Следующая версия Java получила название Java SE 6. Это означает, что в компании Sun вновь решили изменить название платформы Java. Прежде всего, из названия исчезла цифра “2”. Теперь платформа стала называться *Java SE*, официальным именем продукта стало Java Platform, Standard Edition 6, а комплект разработчика приложений получил название JDK 6. Как и цифра “5” в названии J2SE 5, цифра “6” в Java SE 6 означает номер версии программного продукта, тогда как внутренним номером версии является 1.6.

Версия Java SE 6 создавалась на основе платформы J2SE 5, отличаясь от последней рядом нововведений. Изменения в этой версии не такие масштабные, как в предыдущей, но в ней были улучшены библиотеки интерфейса прикладного программирования (API), добавлен ряд новых пакетов и доработана исполняющая среда. По сути, в Java SE 6 были закреплены усовершенствования, внедренные в J2SE 5.

Следующая версия Java получила название Java SE 7, а соответствующий комплект разработчика приложений — JDK 7. Данной версии присвоен внутренний номер 1.7. Java SE 7 — это первая основная версия Java, выпущенная после того, как компания Sun Microsystems была приобретена компанией Oracle. В Java SE 7 появилось немало новых средств, в том числе существенные дополнения были включены как в сам язык, так и в стандартные библиотеки API. Наиболее важные средства, внедренные в Java SE 7, были разработаны в рамках проекта *Project Coin*. Цель проекта заключалась в определении ряд незначительных изменений в языке Java, которые должны были быть внедрены в JDK 7. Вот их краткий перечень:

- возможность управления инструкцией `switch` с помощью объектов класса `String`;
- двоичные целочисленные литералы;
- символы подчеркивания в числовых литералах;
- расширенная инструкция `try`, называемая инструкцией `try` *с ресурсами* и поддерживающая автоматическое управление ресурсами;
- выведение типов (через ромбовидный оператор) при создании обобщенного экземпляра объекта;
- усовершенствованная обработка исключений, благодаря которой несколько исключений могут быть перехвачены в одном (групповом) блоке `catch`, а также улучшенный контроль типов для повторно генерируемых исключений.

Как видите, средства, отнесенные в рамках проекта Project Coin к разряду незначительных языковых изменений, обеспечили преимущества, которые вовсе нельзя считать незначительными. В частности, инструкция `try` с ресурсами оказывает серьезное влияние на написание кода.

Следующая версия Java получила название Java SE 8, а ее комплект разработчика приложений — JDK 8. Внутренний номер версии — 1.8. Комплект JDK 8 существенно расширил возможности языка за счет добавления нового языкового средства — *лямбда-выражений*. Включение в язык лямбда-выражений, изменяющих как концептуальную основу программных решений, так и способ написания кода на Java, будет иметь далеко идущие последствия. Использование лямбда-выражений позволяет упростить исходный код при создании определенных языковых конструкций и уменьшить его объем. Добавление в Java лямбда-выражений привело к появлению в языке нового оператора (->) и нового синтаксического элемента.

Помимо лямбда-выражений в JDK 8 появилось много новых полезных средств. Так, начиная с JDK 8 стало возможным определять реализации по умолчанию для методов, включенных в интерфейсы. Кроме того, в JDK 8 добавлена поддержка JavaFX — новой библиотеки графического интерфейса (GUI). Ожидается, что вскоре JavaFX станет составной частью почти всех Java-приложений и вытеснит технологию Swing в большинстве GUI-проектов. Подводя итог можно сказать, что платформа Java SE 8 оказалась ключевым выпуском, который существенно расширил возможности языка и вынудил пересмотреть подходы к написанию кода на Java.

Следующей версией Java стала Java SE 9. Ее комплект разработчика называется JDK 9, а внутренний номер версии — тоже 9. Это основной выпуск Java, включающий значительные улучшения как самого языка, так и его библиотек. Одна из ключевых новинок, появившихся в JDK 9, — *модули*, позволяющие

устанавливать взаимосвязи и зависимости в коде приложения. Благодаря модулям добавилось еще одно средство контроля доступа. Включение модулей привело к появлению нового элемента синтаксиса, нескольких новых ключевых слов и различных усовершенствований инструментов Java. Модули также оказали колоссальное влияние на библиотеку API, поскольку начиная с версии JDK 9 библиотечные пакеты теперь организованы в виде модулей.

Помимо модулей JDK 9 включает несколько других новинок. Одна из наиболее интересных — интерпретатор JShell, поддерживающий интерактивные эксперименты с кодом. (Об интерпретаторе JShell рассказывается в приложении Г.) Еще одной интересной новинкой стала поддержка закрытых методов интерфейсов. Благодаря этому еще больше расширяется появившаяся в JDK 8 поддержка методов интерфейсов, имеющих реализацию по умолчанию. В JDK 9 добавилась возможность поиска в утилите `javadoc`, дополненная новым тегом `@index` для ее поддержки. Как и в предыдущих версиях, в JDK 9 имеется ряд обновлений и улучшений для библиотек Java API.

Как правило, в каждом очередном выпуске Java именно новые средства привлекают наибольшее внимание. Но нельзя не отметить, что один из ключевых элементов Java был объявлен устаревшим в JDK 9. Речь идет об аплетах. Начиная с JDK 9 задействовать аплеты при создании новых проектов не рекомендуется. Как будет объясняться в главе 1, из-за ухудшения поддержки аплетов браузерами (и ряда других причин) в JDK 9 признан устаревшим весь API аплетов. В настоящее время для развертывания приложений в Интернете рекомендуется использовать Java Web Start. (Краткое описание Java Web Start приведено в приложении В.) В силу того, что аплеты сходят со сцены и не рекомендуются к применению при создании нового кода, они не рассматриваются в книге. Если вы все же хотите познакомиться с ними поближе, обратитесь к предыдущим изданиям книги.

В целом можно сказать, что версия JDK 9 осталась верной инновационным традициям Java. Язык продолжает активно развиваться, сохраняя свою популярность в среде разработчиков и максимально отвечая их потребностям. Предыдущее издание книги было переработано и теперь отражает многочисленные новшества, обновления и дополнения, появившиеся в версии Java SE 9 (JDK 9).

Java SE 10

Весной 2018 года появилась очередная версия языка, Java SE 10 (JDK 10). Она включает ряд интересных новинок, в том числе объявление типов переменных с помощью ключевого слова `var`, объединение “леса” JDK в единый репозиторий, общий доступ к данным класса и многое другое. В приложении Е будут описаны две основные новинки Java SE 10.

Структура книги

Книга представляет собой учебное пособие, разделенное на 17 глав, в каждой из которых рассматривается та или иная тема, связанная с программированием на Java. Каждая последующая глава опирается на материал, изученный в предыдущей главе. Характерной особенностью книги является использование ряда приемов, повышающих эффективность обучения и позволяющих закрепить полученные знания.

Вопросы и упражнения для самопроверки

В конце каждой главы приведены вопросы и упражнения для самопроверки, позволяющие читателю протестировать свои знания. Ответы на вопросы и решения упражнений приведены в приложении А.

Вопросы к эксперту

На страницах книги вам будут встречаться врезки "Спросим у эксперта". Они содержат дополнительные сведения или комментарии к рассматриваемой теме в виде вопросов и ответов.

Упражнения к главам

Каждая глава содержит одно или несколько упражнений, представляющих собой несложные проекты, которые помогут читателям закрепить полученные знания на практике. Как правило, это реальные программы, которые можно использовать в качестве основы для разработки собственных приложений.

Книга для всех программистов

Для чтения книги не требуется предварительный опыт программирования. Конечно, если вы уже программировали ранее, то вам будет проще усваивать материал. Но учитывайте, что Java принципиально отличается от других популярных языков, поэтому не стоит сразу же переходить к упражнениям. Желательно читать книгу последовательно, изучая главы раздел за разделом.

Необходимое программное обеспечение

Для компиляции и запуска программ, исходные коды которых представлены в книге, вам потребуется последняя версия комплекта Java Development Kit (JDK) от компании Oracle. На момент написания книги это был комплект JDK 9 для версии Java SE 9. О том, как найти и установить такой комплект, будет рассказано в главе 1.

Даже если вы пользуетесь более ранней версией Java, то это не помешает вам извлечь пользу из чтения книги. Просто в этом случае вам не удастся скомпилировать и выполнить те программы, в которых используются новые функциональные возможности Java.

Исходный код примеров программ

Все файлы примеров доступны на сайте книги по следующему адресу:

```
https://www.mhprofessional.com/9781259589317-usa-java-a-beginners-
guide-seventh-edition-group
```

Файлы можно также скачать с веб-страницы книги на сайте издательства “Диалектика”:

```
http://www.williamspublishing.com/Books/978-5-6041394-5-5.html
```

Ждем ваших отзывов!

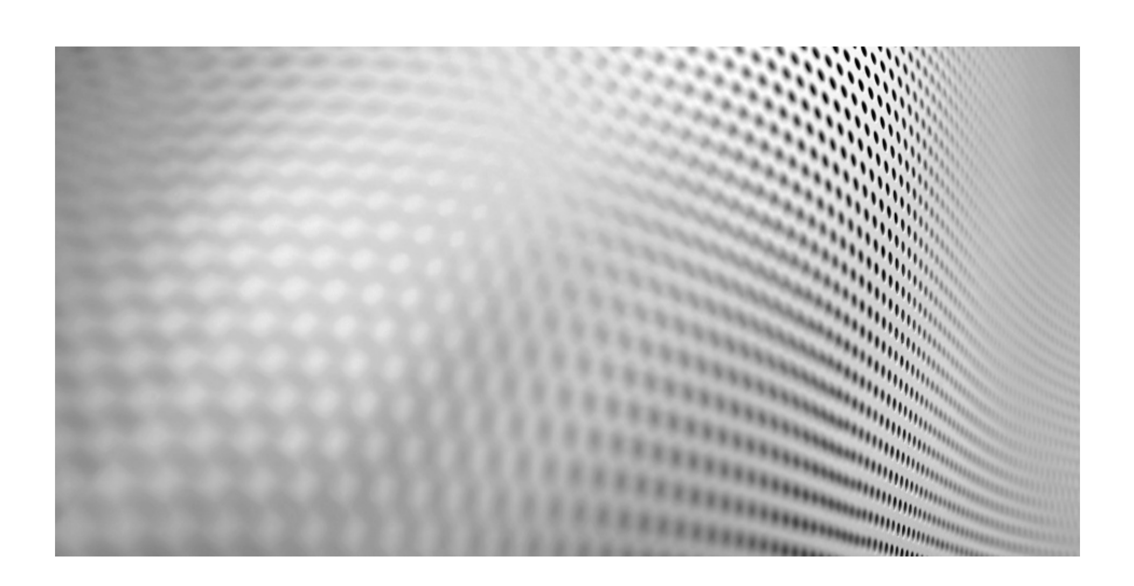
Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@dialektika.com`
WWW: `www.dialektika.com`



Глава 1

Основы Java

В этой главе...

- История развития и основные концепции Java
- Влияние Java на развитие Интернета
- Назначение байт-кода
- Терминология Java
- Основные концепции объектно-ориентированного программирования
- Создание, компиляция и выполнение простой программы на Java
- Использование переменных
- Использование управляющих инструкций `if` и `for`
- Создание блоков кода
- Разбиение на строки, структурирование и завершение инструкций в исходном коде
- Ключевые слова Java
- Правила использования идентификаторов Java

Развитие Интернета и, в частности, Всемирной паутины (WWW) существенно изменило наши представления о сфере применения вычислительной техники. До появления Интернета в распоряжении большинства пользователей были лишь персональные компьютеры, работавшие независимо друг от друга. В настоящее время почти каждый компьютер может иметь доступ к глобальным ресурсам через подключение к Интернету, который, в свою очередь, также подвергся радикальным преобразованиям. Если первоначально Интернет был не более чем удобным средством, позволяющим обмениваться файлами, то теперь он превратился в огромную и разветвленную компьютерную вселенную. Подобные изменения привели к выработке нового подхода к программированию, основанного на языке Java.

Язык Java особенно удобен для написания интернет-приложений, однако это не единственная область его применения. Появление Java стало переломным моментом в программировании, кардинально изменив наши представления о структуре и назначении программ. В настоящее время программист не может считать себя профессионалом, если не умеет писать программы на Java. Освоив материал книги, вы сможете овладеть основами этого языка.

Данная глава служит введением в программирование на Java. Здесь представлена краткая история развития Java, описаны стратегии разработки программ и

наиболее важные средства этого языка. Новичкам труднее всего уяснить, что в любом языке программирования не существует независимых элементов и что все они тесно взаимосвязаны. Подобная взаимосвязь элементов языка характерна и для Java. В самом деле, очень трудно рассматривать какой-то один аспект Java в отрыве от остальных. И для того чтобы преодолеть подобное затруднение, в этой главе дается краткий обзор ряда языковых средств Java, включая общую структуру программы на Java, основные управляющие блоки и конструкции. Не вдаваясь в детали, мы уделим основное внимание общим понятиям и принципам, на основе которых создается любая программа на языке Java.

Истоки Java

Язык Java был задуман в 1991 году сотрудниками компании Sun Microsystems Джеймсом Гослингом, Патриком Нотоном, Крисом Уортом, Эдом Фрэнком и Майком Шериданом. Первоначально он назывался Oak, но в 1995 году, когда за его продвижение взялись маркетологи, он был переименован в Java. Как это ни удивительно, на первых порах сами авторы языка не ставили перед собой задач разработки интернет-приложений. Их целью было создание платформенно-независимого языка, на котором можно было бы писать встраиваемое программное обеспечение для различной бытовой электронной аппаратуры, в том числе тостеров, микроволновых печей и пультов дистанционного управления. Как правило, в устройствах подобного типа применялись контроллеры на базе микропроцессоров различной архитектуры, а исполняемый код, генерируемый компиляторами большинства существовавших в то время языков программирования, был ориентирован на определенные типы процессоров. Характерным тому примером может служить язык C++.

Несмотря на то что программу, написанную на C++, можно выполнить на процессоре практически любого типа, сделать это можно, лишь скомпилировав ее в исполняемый код команд конкретного процессора. Создание компиляторов — длительный и трудоемкий процесс, поэтому в поисках оптимального решения Гослинг и другие члены рабочей группы остановились на межплатформенном языке, для которого компилятор генерировал бы код, способный выполняться на разных процессорах в различных вычислительных средах. В результате их усилия увенчались созданием языка, известного теперь под названием Java.

Пока разработчики Java уточняли детали создаваемого ими языка, началось бурное развитие “Всемирной паутины”, во многом определившей будущее Java. Если бы не формирование веб-сообщества, язык Java, вероятно, нашел бы лишь ограниченное применение, главным образом в разработке программ, встраиваемых в специализированные контроллеры. Но как только широкодоступный Интернет стал реальностью, появилась острая потребность в переносимых программах, что и послужило причиной для выдвижения Java на передний план в качестве основного языка разработки подобных программ.

По мере накопления опыта многие программисты очень быстро приходят к выводу, что переносимость программ — труднодостижимый идеал. Задача создания кросс-платформенных программ возникла едва ли не вместе с появлением первых компьютеров, но взяться за ее решение так и не удавалось из-за необходимости решать другие, более важные и неотложные задачи. Как бы там ни было, но с появлением Интернета проблема переносимости программ перешла в разряд совершенно неотложных. Ведь Интернет состоит из множества разнотипных компьютеров с различной архитектурой процессоров и разными операционными системами.

В итоге увлекательная, но, на первый взгляд, не столь важная задача неожиданно стала чрезвычайно актуальной. В 1993 году разработчикам Java стало ясно, что задачу переносимости нужно решать не только в процессе программирования микропроцессорных устройств, но и при разработке кода для интернет-приложений. Иными словами, сфера применения языка Java внезапно расширилась. И если программирование микроконтроллеров стало побудительной причиной для создания Java, то Интернет способствовал широкому распространению этого языка.

Связь Java с языками C и C++

Java весьма напоминает языки C и C++. От C язык Java унаследовал синтаксис, а от C++ — объектную модель. Сходство Java с языками C и C++ играет важную роль. Во-первых, многие программисты знакомы с синтаксисом C и C++, что упрощает изучение Java. Те же, кто освоил Java, могут без труда изучить C и C++. Во-вторых, тем, кто программирует на Java, не приходится изобретать велосипед. Они могут успешно применять уже известные и хорошо зарекомендовавшие себя подходы. Современная эпоха в программировании, по сути, началась с языка C. Затем появился язык C++, а после него — Java. Имея такое богатое наследство, Java предоставляет программистам высокопроизводительную и удобную среду, в которой реализованы лучшие из уже известных решений и добавлены новые средства, необходимые для интерактивной разработки. Очень важно отметить тот факт, что вследствие своей схожести языки C, C++ и Java сформировали концептуальную основу для профессионального программирования. При переходе от одного языка к другому программистам не приходится преодолевать серьезные трудности, имеющие принципиальный характер.

Один из принципов проектирования, заложенных в основу C и C++, заключается в предоставлении программисту широчайших полномочий. Разработчики Java также следовали этому принципу. Если не учитывать ограничения, накладываемые Интернетом, то следует признать, что Java дает программисту полный контроль над кодом. Если вы умеете грамотно программировать, то это будет видно по вашим программам. Недостаток опыта также отразится на ваших программах. Одним словом, Java — язык не для дилетантов, а для профессионалов.

У Java имеется еще одно сходство с языками C и C++: все эти языки были задуманы, разработаны, проверены и уточнены программистами-практиками. В их основу положены реальные потребности их создателей. При таком подходе к разработке языка программирования велика вероятность получить качественный продукт, способный найти признание у специалистов.

Из-за сходства языков Java и C++, в особенности из-за подобия предоставляемых ими средств для объектно-ориентированного программирования, возникает соблазн рассматривать Java как своего рода версию C++ для Интернета. Но это было бы ошибкой. У Java имеется целый ряд существенных отличий от C++ как в концептуальном, так и в прикладном плане. Несмотря на то что C++ оказал очень сильное влияние на язык Java, последний вовсе не является расширенной версией первого. В частности, эти языки не совместимы ни сверху вниз, ни снизу вверх. Конечно, сходство с языком C++ очень важно, и если у вас имеется опыт программирования на C++, вы будете чувствовать себя в своей стихии, программируя на Java. Но не следует забывать, что язык Java был разработан *не* на замену C++, а для решения вполне определенного круга задач, отличающихся от тех, что решаются с помощью C++. Именно поэтому Java и C++ просто обречены на мирное сосуществование еще долгие годы.

Вклад Java в развитие Интернета

Появление Интернета послужило основной причиной для выхода Java на передовые рубежи программирования. В свою очередь, Java оказал благотворное влияние на развитие Интернета. Этот язык не только упростил веб-программирование, но и положил начало новой разновидности сетевых программ, называемых ***аплетами***, которые полностью изменили представление о том, каким может быть веб-содержимое. Java также позволил решить наиболее сложные задачи, возникающие при создании сетевых программ, а именно: обеспечение переносимости и безопасности.

Спросим у эксперта

ВОПРОС. Что такое C# и как он связан с Java?

ОТВЕТ. Через несколько лет после создания Java корпорация Microsoft разработала похожий язык C#, который тесно связан с Java. У многих языковых средств C# имеются свои аналоги в Java. В Java и C# используется единый общий синтаксис, напоминающий синтаксис C++, поддерживается распределенное программирование и применяется одна и та же объектная модель. Разумеется, у Java и C# имеются отличия, но внешне эти языки очень похожи. Это означает, что, зная C#, вы сможете относительно легко изучить Java, и наоборот: если вам предстоит изучить C#, знание Java может очень пригодиться.

Учитывая сходство Java и C#, вы можете спросить: "Заменит ли C# язык Java?" Безусловно, ответ на этот вопрос будет отрицательным. Java и C# предназначены для абсолютно разных типов вычислительных сред. Как и в случае с языком C++, Java будет мирно сосуществовать с языком C# еще многие годы.

Облегчение разработки интернет-приложений с помощью Java

С появлением Java существенно упростилось программирование интернет-приложений. Возможно, наиболее важное новшество — возможность создавать переносимые кросс-платформенные приложения. Не менее важное значение имеет поддержка сетей со стороны Java. Благодаря наличию библиотеки, включающей набор готовых функций, программисты могут легко создавать приложения, взаимодействующие с Интернетом. Также в Java поддерживаются механизмы, которые позволяют легко доставлять программы через Интернет. Не вдаваясь в подробности, выходящие за рамки книги, отметим, что поддержка сетевых вычислений в Java является ключевым фактором быстрого роста популярности этого языка.

Java-аплеты

Аплет — это особая разновидность программ на Java, предназначенная для передачи через Интернет и автоматического выполнения в среде, формируемой Java-совместимым браузером. Аплет загружается по требованию клиентской программы, а для его передачи по сети вмешательство пользователя не требуется. Если пользователь щелкает на ссылке, которая указывает на документ, содержащий аплет, последний будет автоматически скопирован и запущен браузером. Большинство аплетов имеет небольшие размеры. Обычно они служат для отображения информации, предоставляемой серверами, или для поддержки ввода данных пользователем. Иногда с их помощью реализуются несложные функции. Например, кредитный калькулятор удобнее разместить в виде аплета на стороне клиента, чем выполнять вычисления на стороне сервера. Таким образом, аплет позволяет переносить некоторые функции с сервера на компьютер клиента.

С появлением аплетов расширился круг объектов, пригодных для свободной передачи в сетевой среде. Существуют две категории объектов, которыми сервер может обмениваться с клиентом: пассивные (статические) данные и активные исполняемые программы (динамические данные). Например, просматривая электронную почту, вы имеете дело со статическими данными. Даже если в почтовом отправлении пересылается программа, то ее код не активизируется до тех пор, пока не получит управление. Аплет, напротив, является динамической, самостоятельно выполняющейся программой, для запуска которой не

приходится принимать никаких мер. Такие программы играют роль активных агентов на клиентских машинах, но инициализируются сервером.

На заре Java аплеты были важной частью программирования на Java. Они иллюстрировали мощь и преимущества Java, добавили "третье измерение" на веб-страницы и обеспечили программистам доступ ко всему спектру возможностей Java. Несмотря на то что аплеты используются до сих пор, их значимость уже не столь высока. Как будет рассказано в следующих разделах, начиная с JDK 9 аплеты постепенно уходят со сцены. На смену им приходят другие механизмы, которые обеспечивают альтернативный способ доставки динамических программ в Интернете.

Безопасность

Использование динамических сетевых программ может вызывать серьезные проблемы с точки зрения безопасности и переносимости. Очевидно, что следует предпринять меры по предотвращению возможного вреда со стороны программ, которые загружаются и автоматически выполняются на клиентском компьютере. Эти программы должны также выполняться в самых разных средах и операционных системах. Как будет показано далее, проблемы, возникающие при выполнении Java-приложений, решаются эффективным и элегантным образом. Рассмотрим подробнее способы решения этих проблем, начиная с проблем, связанных с безопасностью.

Как известно, запуск обычной программы, загруженной через Интернет, сопряжен с риском, поскольку она может быть заражена вирусом или служить своего рода "троянским конем", предназначенным для злонамеренного проникновения в систему. А злонамеренные действия такой программы возможны из-за того, что она получает несанкционированный доступ к системным ресурсам. Так, вирус, анализируя содержимое файловой системы локального компьютера, может собирать конфиденциальные данные, например номера банковских карт, сведения о банковских счетах и пароли. Для безопасной загрузки и запуска программ на клиентской машине необходимо устранить саму возможность атаки на систему со стороны этих программ.

Защита от атак реализуется путем создания специальной среды для выполнения приложения, не позволяющей ему обращаться к ресурсам компьютера (далее будет показано, как решается подобная задача). Возможность загружать приложение с уверенностью в том, что оно не нанесет вреда системе, относится к числу наиболее привлекательных особенностей Java.

Переносимость

Переносимость является важным свойством сетевых программ. Значимость этой характеристики обусловлена тем, что в сети могут присутствовать разнотипные компьютеры, работающие под управлением различных операционных систем. Если программа на Java предназначена для выполнения на

произвольном компьютере, подключенном к Интернету, то должны существовать способы обеспечения работы этой программы в различных системах. Например, одно и то же приложение должно выполняться на компьютерах с разнотипными процессорами, в разных операционных системах и с различными браузерами. Хранить разные версии приложения для разнотипных компьютеров слишком сложно, если вообще возможно. *Один и тот же* код должен выполняться на *всех* компьютерах. Таким образом, необходима поддержка процесса генерации переносимого исполняемого кода. Как станет ясно в дальнейшем, те же самые средства, которые обеспечивают безопасность, помогают добиться и переносимости программ.

Волшебный байт-код Java

Добиться безопасности и переносимости сетевых программ позволяет генерируемый компилятором Java код, не являющийся исполняемым. Такой код называется *байт-кодом*. Это оптимизированный набор команд, предназначенных для выполнения в исполняющей среде, называемой *виртуальной машиной Java* (Java Virtual Machine — JVM). Виртуальная машина Java фактически представляет собой *интерпретатор байт-кода*. Такой подход может показаться не совсем обычным, поскольку для повышения производительности большинства современных языков применяются компиляторы, генерирующие исполняемый код. Но выполнение программы под управлением виртуальной машины позволяет разрешить многие трудности, возникающие в работе веб-приложений.

Трансляция исходного кода Java в байт-код существенно упрощает перенос программ из одной среды в другую, поскольку для обеспечения работоспособности кода достаточно реализовать на каждой платформе виртуальную машину. Если на компьютере имеется пакет исполняющей среды, то на нем может выполняться любая программа, написанная на Java. Несмотря на то что виртуальные машины на различных платформах могут быть реализованы по-разному, они должны одинаково интерпретировать байт-код. Если бы исходный текст программы на Java компилировался в собственный код, для каждого типа процессора, взаимодействующего с Интернетом, необходимо было бы предусмотреть отдельную версию данной программы. Такое решение нельзя назвать приемлемым. Следовательно, выполнение байт-кода под управлением виртуальной машины — самый простой путь к обеспечению переносимости программ.

Выполнение программы под управлением виртуальной машины помогает также обеспечить безопасность. Виртуальная машина может запретить программе выполнять операции, побочные эффекты которых способны повлиять на ресурсы за пределами исполняющей среды. Кроме того, безопасность достигается посредством наложения некоторых ограничений, предусмотренных в языке Java.

Как правило, интерпретируемая программа выполняется медленнее, чем скомпилированная в машинный код. Но для кода Java отличия в быстродействии не очень существенны. Ведь байт-код оптимизирован, и поэтому программа выполняется под управлением виртуальной машины значительно быстрее, чем следовало бы ожидать.

Несмотря на то что Java был задуман как интерпретируемый язык, ничто не мешает использовать оперативную (т.е. выполняемую на лету) компиляцию байт-кода в собственный код процессора для повышения производительности. С этой целью сразу же после первой реализации JVM компания Sun Microsystems начала работу над технологией HotSpot, в рамках которой был разработан динамический компилятор байт-кода. Если в состав виртуальной машины входит динамический компилятор, то байт-код по частям преобразуется в собственный исполняемый код. Преобразовывать сразу всю программу на Java в исполняемый код нецелесообразно из-за разнообразных проверок, которые могут производиться только на этапе выполнения программы. Поэтому динамический компилятор осуществляет преобразования кода частями по мере необходимости. Отсюда его другое название — JIT (Just In Time), т.е. компилятор, вступающий в действие лишь в нужный момент времени. Более того, компиляции подвергаются не все фрагменты байт-кода, а лишь те, скорость выполнения которых можно повысить благодаря компиляции, а остальной код интерпретируется. Несмотря на все ограничения, присущие динамической компиляции, она тем не менее позволяет существенно повысить производительность программ. И невзирая на динамическое преобразование байт-кода в исполняемый код, переносимость и безопасность сохраняются, поскольку JVM по-прежнему участвует в процессе выполнения программ.

А теперь еще один момент. Начиная с JDK 9 отдельные среды Java также включают *ранний* компилятор, который может быть использован для компиляции байт-кода в платформенно-ориентированный (машинный) код *до* выполнения JVM, а не на лету. Ранняя компиляция — это специализированное средство, которое не заменяет только что описанный традиционный подход, применяемый в Java. Более того, ранней компиляции присущ ряд ограничений. На момент написания книги ранняя компиляция применялась только в экспериментальных целях, была доступна лишь для 64-разрядных версий Java на платформе Linux, а предварительно скомпилированный код должен выполняться на той же (или на похожим образом сконфигурированной) системе, на которой был скомпилирован код. Это приводит к тому, что ранняя компиляция снижает степень переносимости. В силу узкоспециализированного характера ранней компиляции она не будет подробно рассматриваться в данной книге.

Спросим у эксперта

ВОПРОС. Мне приходилось слышать о специальных программах на Java, называемых сервлетами. Что это такое?

ОТВЕТ. *Сервлет* — это небольшая программа, выполняемая на стороне сервера. Сервлеты динамически реализуют дополнительные функции веб-сервера. Как бы ни были важны клиентские приложения, они реализуют только одну часть архитектуры "клиент — сервер". Прошло совсем немного времени с момента появления Java, как специалистам стало ясно, что этот язык может оказаться полезным и на стороне сервера. В результате появились сервлеты. Благодаря сервлетам процесс взаимодействия клиентов и серверов может полностью поддерживаться в программах на Java. Вопросы функционирования сервлетов выходят за рамки книги для начинающих, но в дальнейшем вам будет полезно изучить этот тип программ. Более подробно о сервлетах можно узнать из книги *Java. Полное руководство, 10-е издание*.

За пределами аплетов

Как уже упоминалось ранее, сразу же после появления Java аплеты играли критически важную роль в разработке приложений. Благодаря им "ожили" веб-страницы, и аплеты стали наиболее заметной частью Java. Но проблема заключается в том, что аплеты используют подключаемый Java-модуль браузера. Поэтому для работы аплета нужна поддержка со стороны браузера. Однако в последних версиях браузеров поддержка подключаемого модуля Java ухудшилась. Без поддержки со стороны браузера аплеты просто нежизнеспособны. Поэтому начиная с версии JDK 9 поддержка аплетов Java относится к категории не рекомендуемых. В языке Java категория "не рекомендуемый" означает, что соответствующее средство все еще доступно, но помечено как устаревшее. Не рекомендуемое средство может быть устранено в будущих выпусках языка. Поэтому подобные средства нежелательно использовать в новом коде.

Существуют различные альтернативы аплетам, и, возможно, наиболее важной из них является Java Web Start. Благодаря Java Web Start приложение может динамически загружаться с веб-страницы. Особенность заключается в том, что приложение выполняется само по себе, а не в браузере, не полагаясь на подключаемый модуль Java. Механизм развертывания Java Web Start может работать со многими типами программ на Java. И хотя рассмотрение стратегий развертывания выходит за рамки данной книги, в силу их важности краткое введение в Java Web Start будет представлено в приложении В.

Основные характеристики Java

Даже самый краткий обзор языка Java будет неполным без упоминания его основных свойств. И хотя главной причиной, побудившей к разработке Java, стала потребность в языке, обеспечивающем переносимость и безопасность программ, заметное влияние на оформление Java в окончательном виде оказали и другие факторы. Ниже вкратце перечислены основные характеристики этого языка программирования.

Простота	Java обладает лаконичными, тесно связанными друг с другом языковыми средствами, которые легко изучать и применят
Безопасность	Java предоставляет безопасные средства для создания интернет-приложений
Переносимость	Программы на Java могут выполняться в любой среде, для которой имеется исполняющая подсистема Java
Объектно-ориентированный характер	В Java воплощена современная философия объектно-ориентированного программирования
Надежность	Java уменьшает вероятность появления ошибок в программах благодаря строгой типизации переменных и выполнению соответствующих проверок во время выполнения
Многопоточность	Java обеспечивает встроенную поддержку многопоточного программирования
Архитектурная независимость	Язык Java не привязан к конкретному типу вычислительной среды или архитектуре операционной системы
Интерпретируемость	Java предоставляет байт-код, обеспечивающий независимость от платформы
Высокая производительность	Байт-код Java максимально оптимизируется для повышения производительности
Распределенность	Язык Java проектировался с учетом его применения в распределенной среде Интернета
Динамичность	Программы на Java содержат большой объем информации о типах данных, используемой во время выполнения для предоставления доступа к объектам

Объектно-ориентированное программирование

Одним из главных свойств Java является поддержка объектно-ориентированного программирования (ООП). Объектная методология неотделима от Java, а все программы на Java в той или иной степени являются объектно-ориентированными. Поэтому имеет смысл кратко рассмотреть принципы ООП, прежде

чем переходить к написанию даже самой простой программы на Java. Далее вы увидите, как эти принципы реализуются на практике.

Объектно-ориентированный подход к программированию позволяет разрабатывать достаточно сложные программы. С момента появления первого компьютера методология программирования претерпела ряд существенных изменений, связанных с ростом сложности программ. На заре вычислительной техники процесс программирования представлял собой ввод машинных команд в двоичной форме с пульта управления ЭВМ. В то время размеры программ не превышали нескольких сотен команд, и поэтому такой подход считался вполне приемлемым. Затем появились языки ассемблера. Символьное представление машинных команд и процедура компиляции позволили перейти к созданию более сложных программ. В связи с дальнейшим увеличением объема программного кода появились языки высокого уровня. Они стали теми инструментами, которые позволили программистам справиться с постепенным усложнением программ. Первым из широко распространенных языков высокого уровня стал FORTRAN. Появление FORTRAN стало важным этапом в развитии языков программирования, но этот язык не вполне подходил для создания удобочитаемых программ.

В 1960-е годы начало зарождаться структурное программирование. Впоследствии для поддержки данного подхода были разработаны такие языки, как С и Pascal. Благодаря структурированным языкам программирования появилась возможность относительно легко создавать программы средней сложности. Главными свойствами структурированных языков стали поддержка подпрограмм, локальных переменных, наличие расширенного набора управляющих конструкций и отсутствие оператора GOTO. Но, несмотря на то что структурированные языки стали мощными инструментами программирования, по мере роста объема и сложности проектов их возможности быстро исчерпались.

На каждом очередном этапе развития методологии и инструментальных средств программирования разработчики получали возможность создавать все более сложные программы. На этом пути очередной подход наследовал лучшие черты своих предшественников, а кроме того, приобретал новые качества, позволявшие двигаться вперед. К моменту разработки принципов ООП многие проекты стали настолько сложными, что управлять ими средствами структурного программирования уже не представлялось возможным. Объектно-ориентированная методология позволила разработчикам преодолеть эти препятствия.

Объектно-ориентированное программирование переняло лучшие идеи структурного программирования, дополнив их новыми понятиями. В результате возник новый способ организации программ. В принципе, программы могут создаваться двумя путями: на основе кода (выполняющего действия) и на основе данных (подвергающихся обработке). При использовании только принципов структурного программирования программы организуются на основе кода. Такой подход можно рассматривать как код, воздействующий на данные.

Объектно-ориентированное программирование подразумевает другой подход. Программы организуются на основе данных по следующему главному принципу: данные управляют доступом к коду. В объектно-ориентированных языках программирования определяются данные и процедуры, которым разрешается обрабатывать эти данные. Таким образом, тип данных определяет те операции, которые применимы к этим данным.

Во всех объектно-ориентированных языках программирования, в том числе и в Java, поддерживаются три основных принципа ООП: инкапсуляция, полиморфизм и наследование. Рассмотрим каждый из них по отдельности.

Инкапсуляция

Инкапсуляция представляет собой механизм программирования, объединяющий код и данные, которыми манипулирует этот код, что позволяет предотвратить несанкционированный доступ к данным извне и их некорректное использование. В объектно-ориентированных языках программирования код и данные организуются в некое подобие *черного ящика*. В результате такого объединения создается объект. Иными словами, объект — это компонент, поддерживающий инкапсуляцию.

Данные и код внутри объекта могут быть *закрытыми* (private) или *открытыми* (public). Закрытый код или данные доступны только элементам, содержащимся в том же самом объекте. Поэтому обратиться к такому коду или данным извне объекта невозможно. Если код или данные являются открытыми, то к ним можно обращаться из любой части программы (несмотря на то, что они находятся внутри объекта). Как правило, открытые элементы объекта используются для создания управляемого интерфейса к его закрытым элементам.

Основной языковой конструкцией, поддерживающей инкапсуляцию в Java, является *класс*. Классы будут подробнее рассматриваться далее, но о них нужно сказать несколько слов уже сейчас. Класс определяет тип объекта. В нем описываются как данные, так и код, выполняющий те или иные действия над этими данными. В Java определение, или так называемая спецификация, класса служит для построения объектов. Объекты представляют собой *экземпляры классов*. Следовательно, класс — это ряд "чертежей", по которым строится объект.

Код и данные в составе класса называются *членами* класса. Данные, определенные внутри класса, принято называть *переменными-членами*, или *переменными экземпляра*, а код, выполняющий действия над этими данными, — *методами-членами*, или просто *методами*. *Метод* — это термин, которым в Java принято обозначать подпрограмму. Если вы знакомы с языками C/C++, то, вероятно, знаете, что в этих языках для той же самой цели служит термин *функция*.

Полиморфизм

Полиморфизм (от греческого слова, означающего "много форм") — это свойство, позволяющее с помощью одного интерфейса обращаться к общему классу

действий. Конкретное действие определяется ситуацией. В качестве примера, позволяющего лучше понять принцип полиморфизма, можно привести руль автомобиля. Руль (т.е. интерфейс взаимодействия) остается одним и тем же независимо от того, какой именно механизм рулевого управления применяется в автомобиле, будь то зубчатая, реечная передача или гидроусилитель. Таким образом, зная, как обращаться с рулем, вы сможете управлять автомобилем любого типа.

Тот же самый принцип применяется и в программировании. Рассмотрим в качестве примера стек (структуру данных, организованных по принципу "последним поступил — первым обслужен"). Допустим, в программе требуются три разных стека. Первый стек служит для хранения целочисленных значений, второй — для хранения значений с плавающей точкой и третий — для хранения символьных значений. Каждый стек реализуется с помощью одного и того же алгоритма, несмотря на то, что в стеках хранятся разнотипные данные. В случае языка, не поддерживающего ООП, пришлось бы создавать три разных набора процедур управления стеками, присвоив им разные имена. Но в Java благодаря полиморфизму можно создать один общий набор процедур управления стеками, который будет действовать по-разному в зависимости от конкретного типа стека. Таким образом, зная, как работать с одним стеком, можно взаимодействовать со всеми тремя стеками.

Принцип полиморфизма хорошо иллюстрируется следующим выражением: "один интерфейс — множество методов". Это означает возможность создания универсального интерфейса для группы взаимосвязанных действий. Полиморфизм упрощает программу благодаря возможности определить *общий класс действий* с помощью одного и того же интерфейса. Выбрать определенное действие (т.е. метод) — задача компилятора, и он решает ее в зависимости от конкретных условий. Как программисту вам не приходится выбирать метод вручную. Нужно лишь помнить принципы использования общего интерфейса.

Наследование

Наследование — это процесс, в ходе которого один объект приобретает свойства другого объекта. Наследование имеет очень важное значение, поскольку с его помощью поддерживается иерархическая классификация. Если вдуматься, то знания чаще всего имеют иерархическую структуру. Например, яблоки конкретного сорта относятся к классу *яблок*, который в свою очередь относится к классу *фруктов*, а тот — к более обширному классу *продуктов питания*. В данном случае *продукты питания* обладают определенными свойствами (они съедобны, калорийны и т.п.). Эти же свойства автоматически присущи подклассу *фруктов*. Кроме того, класс *фруктов* обладает дополнительными свойствами (сочные, сладкие и т.п.), что отличает его от классов других продуктов питания. *Яблоки* имеют более конкретные свойства (растут на деревьях, не являются тропическими плодами и т.п.) И наконец, отдельный сорт яблок наследует все

свойства описанных выше классов и, кроме того, обладает особыми свойствами, отличающими его от других сортов яблок.

Без такой иерархической структуры для каждого объекта пришлось бы заново определять весь набор свойств. Благодаря наследованию для объекта достаточно указать те свойства, которые отличают его от других классов, а остальные общие атрибуты он наследует от своих родительских классов. Таким образом, благодаря наследованию можно создать объект, являющийся экземпляром более общего класса.

Установка Java Development Kit

Итак, усвоив теоретические основы Java, можно приступать к написанию программ на Java. Для того чтобы написанную программу можно было скомпилировать и выполнить, вам нужно установить на своем компьютере пакет Java Development Kit (JDK), поддерживающий процесс разработки программ на Java. Этот комплект свободно предоставляется компанией Oracle. На момент написания книги текущей являлась версия JDK 10, применяемая на платформе Java SE 10, где SE — аббревиатура от Standard Edition (стандартный выпуск). Комплект JDK 10 содержит немало новых средств, которые не поддерживаются в предыдущих версиях Java. Для компиляции и выполнения примеров программ, представленных в книге, рекомендуется использовать JDK 9 или более позднюю версию. В случае использования более ранних версий программы, содержащие новые средства, не смогут быть скомпилированы.

JDK можно загрузить по следующему адресу:

`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

Достаточно перейти на страницу загрузки по указанному адресу и следовать инструкциям для вашего типа компьютера и операционной системы. После установки JDK вы сможете компилировать и запускать программы. Среди многих программ, входящих в состав JDK, вам в первую очередь понадобятся две: `javac` и `java`. Первая из них — это компилятор Java, а вторая — стандартный интерпретатор Java, который также называют *модулем запуска приложений*.

Следует, однако, иметь в виду, что программы, входящие в состав JDK, запускаются из командной строки. Они не являются оконными приложениями и не составляют интегрированную среду разработки (IDE).

Примечание

Помимо основных инструментальных средств JDK, доступных из командной строки, для разработки программ на Java имеются интегрированные среды разработки наподобие NetBeans и Eclipse. Интегрированная среда может оказаться очень удобной для разработки и развертывания коммерческих приложений. Как правило, в интегрированной среде можно легко скомпилировать и выполнить примеры программ, представленных в данной книге. Но инструкции по компиляции и выполнению примеров программ

приводятся в книге только для инструментальных средств JDK, доступных из командной строки. Это сделано по ряду причин. Во-первых, комплект JDK доступен для всех читателей книги. Во-вторых, инструкции по применению JDK одинаковы для всех пользователей. А кроме того, простые примеры программ, представленные в книге, проще всего компилировать и выполнять инструментальными средствами JDK из командной строки. Если же вы пользуетесь интегрированной средой разработки, то вам придется следовать ее инструкциям. А в силу отличий, имеющихся у разных интегрированных сред разработки, дать общие инструкции по компиляции и выполнению программ, написанных на Java, не представляется возможным.

Спросим у эксперта

ВОПРОС. Вы утверждаете, что объектно-ориентированное программирование — эффективный способ управления крупными программами. Но в случае небольших программ отрицательное влияние накладных расходов, связанных с применением ООП, на производительность будет, вероятно, относительно более заметным. Все программы на Java в той или иной степени являются объектно-ориентированными. Означает ли это, что данный язык неэффективен для написания небольших программ?

ОТВЕТ. Нет, не означает. Как будет показано далее, даже в случае небольших программ применение методик ООП почти не сказывается на производительности. Несмотря на то что язык Java строго придерживается объектной модели, окончательное решение о том, в какой степени применять его объектно-ориентированные средства, остается за вами. Для небольших программ объектно-ориентированный подход вряд ли можно считать целесообразным. Но по мере того, как сложность ваших программ будет увеличиваться, вы сможете без труда применять более развитые средства ООП.

Первая программа на Java

Начнем с компиляции и запуска простой программы.

```
/*
   Это пример простой программы на Java.

   Присвойте файлу с исходным кодом имя Example.java.
*/
class Example {
   // Выполнение программы на Java начинается с вызова метода main()
   public static void main(String args[]) {
       System.out.println("Java правит Интернетом!");
   }
}
```

Итак, вам предстоит выполнить следующие действия:

1) ввести исходный код программы;
2) скомпилировать программу;
3) запустить программу на выполнение.

Ввод исходного кода программ

Исходные коды примеров программ, представленных в книге, доступны для загрузки на сайте книги, адрес которого был указан во введении, но по желанию можно вводить код вручную. В таком случае следует использовать обычные текстовые редакторы, а не текстовые процессоры вроде Word, записывающие в файл не только текст, но и данные о его форматировании, которые будут восприняты компилятором как недопустимые языковые конструкции. Если вы работаете на платформе Windows, вам вполне подойдет WordPad или другой простой текстовый редактор.

В большинстве языков программирования допускается присваивать произвольное имя файлу, содержащему исходный код программы. Но в Java действуют иные правила. Для программирования на Java следует знать, что *имя исходного файла играет очень важную роль.* В рассматриваемом здесь примере файлу, содержащему исходный код программы, нужно присвоить имя `Example.java`. Ниже поясняются причины, по которым выбирается именно такое имя файла.

В Java файл с исходным кодом формально называется *единицей компиляции.* Это текстовый файл, содержащий определения одного или нескольких классов. (Исходные файлы первых примеров программ будут содержать только один класс.) Компилятор Java требует, чтобы исходный файл имел расширение `.java`. Анализируя исходный код первого примера программы, вы заметите, что класс тоже называется `Example`. И это не случайно. В программах на Java весь код должен находиться в классе. Согласно принятым соглашениям, имя файла, содержащего исходный текст программы, должно совпадать с именем класса. Нетрудно убедиться, что имя файла в точности соответствует имени класса вплоть до регистра. Дело в том, что в Java имена и другие идентификаторы зависят от регистра символов. На первый взгляд, условие соответствия имен классов и файлов может показаться слишком строгим, но оно упрощает организацию и сопровождение программ, как станет ясно из последующих примеров.

Компиляция программы

Для компиляции программы `Example` запустите компилятор `javac`, указав в командной строке имя исходного файла:

```
C:\>javac Example.java
```

Компилятор `javac` создаст файл `Example.class`, содержащий байт-код программы. Напомним, что байт-код не является исполняемым, но интерпретируется виртуальной машиной Java. Таким образом, результат компиляции в `javac` нельзя запускать на выполнение непосредственно.

Для запуска скомпилированной программы следует воспользоваться интерпретатором `java`. В качестве параметра ему нужно передать имя класса `Example` в командной строке:

```
C:\>java Example
```

В результате выполнения программы на экран будет выведена следующая строка:

```
Java правит Интернетом!
```

При компиляции исходного кода Java каждый класс помещается в отдельный выходной файл, называемый по имени класса и получающий расширение `.class`. Именно по этой причине и выдвигается требование, чтобы имя исходного файла программы на Java в точности соответствовало имени содержащегося в нем класса, а по сути — имени файла с расширением `.class`. При вызове интерпретатора Java ему передается имя класса, предназначенного для выполнения (пример такого вызова приведен выше). В результате интерпретатор автоматически находит файл с указанным именем и расширением `.class`. Обнаружив этот файл, интерпретатор выполняет код, указанный в классе.

Примечание

Если при попытке скомпилировать программу компилятор `javac` не будет найден, вам придется указать полный путь к инструментальным средствам JDK, доступным из командной строки, при условии, что комплект JDK установлен правильно. В Windows это, например, означает, что вам нужно ввести путь к доступным из командной строки инструментальным средствам JDK, указанный в переменной среды `PATH`. Так, если комплект JDK 10 установлен в выбираемых по умолчанию каталогах, путь к его инструментальным средствам будет следующим: `C:\Program Files\Java\jdk-10\bin`. (Разумеется, если комплект JDK 10 установлен в другом каталоге или вы используете другую его версию, этот путь нужно будет изменить.) Чтобы узнать, как задавать путь к файлам, обратитесь к справке конкретной операционной системы, поскольку в разных операционных системах эта процедура может отличаться.

Построчный анализ исходного кода примера

Несмотря на то что программа `Example.java` очень проста и объем ее кода невелик, она обладает рядом свойств, общих для всех программ на Java. Проанализируем эту программу по частям.

Ее исходный код начинается так.

```
/*
   Это пример простой программы на Java.

   Присвойте файлу с исходным кодом имя Example.java.
*/
```

Это *комментарии*. Как и в большинстве других языков программирования, в Java разрешается комментировать исходный текст программы. Комментарии игнорируются компилятором. Текст комментариев содержит описание работы программы, предназначенное для тех, кто будет просматривать ее исходный код. В данном случае комментарии сообщают, что программа проста, а ее исходному файлу следует присвоить имя Example.java. Очевидно, что в реальных приложениях комментарии должны описывать, какие именно функции выполняют отдельные фрагменты программы или зачем используются те или иные языковые конструкции.

В Java поддерживаются три вида комментариев. Первый из них находится в начале рассматриваемой здесь программы и позволяет задавать *многострочные комментарии*. Они начинаются символами /* и оканчиваются символами */. Любое содержимое, находящееся между этими ограничителями, игнорируется компилятором.

Далее следует такая строка кода:

```
class Example {
```

В ней имеется ключевое слово class, с помощью которого определяется новый класс. Как упоминалось ранее, класс является основной языковой конструкцией Java, поддерживающей инкапсуляцию, и Example — это имя класса. Определение класса начинается открывающей фигурной скобкой ({) и заканчивается закрывающей фигурной скобкой (}). Элементы, находящиеся между ними, являются членами класса. Не пытайтесь пока что разобраться в особенностях классов, но имейте в виду, что любой код, какие бы действия он ни выполнял, находится внутри класса. Такая организация свидетельствует о том, что любая программа на Java является в той или иной мере объектно-ориентированной.

Следующая строка кода содержит *однострочные комментарии*.

```
// Выполнение программы на Java начинается с вызова метода main()
```

Это второй вид комментариев, поддерживаемых в Java. Однострочные комментарии начинаются с символов // и продолжаются до конца строки. Как правило, многострочные комментарии служат для длинных примечаний, а однострочные — для коротких заметок.

Следующая анализируемая строка кода выглядит так:

```
public static void main (String args[]) {
```

В этой строке определяется метод main(). Как упоминалось ранее, в терминологии Java подпрограммы принято называть *методами*. Именно с данной строки начинается работа программы, и приведенные выше комментарии подтверждают это. Выполнение любой программы на Java начинается с вызова метода main(). Мы не будем пока что касаться назначения каждого элемента анализируемой строки кода. Ведь для этого нужно сначала рассмотреть ряд других языковых средств Java. Но поскольку многие примеры программ,

представленные в книге, содержат строку с подобным определением метода `main()`, рассмотрим вкратце ее составные части.

Ключевое слово `public` называется ***модификатором доступа***. Модификатор доступа определяет правила обращения к членам класса из других частей программы. Если член класса предваряется ключевым словом `public`, то к нему можно обращаться за пределами класса. (В отличие от `public`, модификатор доступа `private` запрещает доступ к членам класса за его пределами.) В данном случае метод `main()` должен быть объявлен как `public`, поскольку при выполнении программы он вызывается за пределами своего класса. Ключевое слово `static` допускает вызов метода `main()` до создания объекта класса. Указывать его необходимо, поскольку метод `main()` вызывается виртуальной машиной еще до того, как будут созданы какие-либо объекты. Ключевое слово `void` лишь сообщает компилятору о том, что метод `main()` не возвращает никаких значений. Как будет показано далее, для некоторых методов предусматривается возвращаемое значение. Если вам пока еще не все понятно в анализируемой строке кода, не отчаивайтесь. Упомянутые здесь языковые средства Java будут подробно рассмотрены в последующих главах.

Как упоминалось выше, метод `main()` вызывается в начале выполнения программы на Java. Любые данные, которые требуется передать этому методу, задаются с помощью переменных, указываемых в круглых скобках после имени метода. Эти переменные называются ***параметрами***. Если для какого-нибудь метода параметры не предусмотрены, то после его имени указывается лишь пара круглых скобок. В данном случае для метода `main()` под именем `args` задается единственный параметр `String args[]` — это массив объектов типа `String`. (Массивы представляют собой наборы однотипных объектов.) В объектах типа `String` хранятся последовательности символов. В данном случае в массиве `args` методу `main()` передаются в виде аргументов параметры, указываемые в командной строке при запуске программы. В анализируемой здесь программе данные, получаемые из командной строки, не используются, но в других примерах программ будет показано, каким образом можно обработать эти данные.

И завершается анализируемая строка кода символом `{`, обозначающим начало тела метода `main()`. Весь код, содержащийся в методе, располагается между открывающей и закрывающей фигурными скобками.

Очередная анализируемая строка кода приведена ниже. Обратите внимание на то, что она содержится внутри метода `main()`.

```
System.out.println("Java правит Интернетом!");
```

В этой строке кода на экран сначала выводится символьная строка "Java правит Интернетом!", а затем выполняется переход на новую строку. Вывод на экран осуществляется встроенным методом `println()`. В данном случае метод `println()` выводит на экран переданную ему символьную строку. Как будет показано далее, с помощью метода `println()` можно выводить на экран

не только символьные строки, но и данные других типов. Анализируемая строка кода начинается с выражения `System.out`. В настоящий момент объяснить его назначение нелегко, поэтому достаточно будет сказать, что `System` — это предопределенный класс, предоставляющий доступ к системным ресурсам, а `out` — поток вывода на консоль. Таким образом, `System.out` — это объект, инкапсулирующий вывод на консоль. Тот факт, что для определения консоли в Java используется объект, лишний раз свидетельствует об объектно-ориентированном характере этого языка.

Как вы, вероятно, уже догадались, в реальных программах на Java вывод на консоль (как, впрочем, и ввод с консоли) используется очень редко. Современные вычислительные среды, как правило, имеют оконный и графический интерфейс, и поэтому обмен данными с консолью применяется лишь в простых служебных и демонстрационных программах. Далее будут рассмотрены другие способы формирования выходных данных средствами Java, а пока ограничимся использованием методов ввода-вывода на консоль в представленных примерах программ.

Обратите внимание на то, что строка, содержащая вызов метода `println()`, оканчивается точкой с запятой. Этим символом завершаются все инструкции в Java. В других строках кода точка с запятой отсутствует лишь потому, что они формально не являются инструкциями.

Первая закрывающая фигурная скобка завершает метод `main()`, а вторая такая же скобка указывает на завершение определения класса `Example`.

И последнее замечание: в Java учитывается регистр символов. Игнорирование этого правила влечет за собой серьезные проблемы. Так, если вместо `main` вы случайно наберете `Main`, а вместо `println` — `PrintLn`, то исходный код программы окажется некорректным. И хотя компилятор Java скомпилирует классы, не содержащие метод `main()`, у него не будет средств, применяемых для их выполнения. Следовательно, если вы допустите ошибку в имени `main`, то компилятор скомпилирует программу и не сообщит об ошибке. Но о ней вас уведомит интерпретатор, когда не сумеет обнаружить метод `main()`.

Обработка синтаксических ошибок

Введите текст рассмотренной выше программы, скомпилируйте и запустите ее (если вы не сделали этого раньше). Если у вас имеется определенный опыт программирования, то вам, безусловно, известно о том, что никто не застрахован от ошибок при наборе исходного кода программы, а также о том, насколько легко допустить такие ошибки. Если вы наберете исходный код программы неправильно, то компилятор, пытаясь обработать этот код, выведет *сообщение об ошибке*. Компилятор Java следует формальным правилам, и поэтому строка в исходном коде, на которую он указывает, не всегда соответствует истинному местоположению источника ошибки. Допустим, в рассмотренной ранее

программе вы забыли ввести открывающую фигурную скобку после метода main(). В таком случае компилятор сообщит вам о следующих ошибках.

```
Example.java:8: ';' expected
   public static void main(String args[])
                                         ^
Example.java:11: class, interface, or enum expected
}
^
```

Очевидно, что первое сообщение не соответствует действительности, так как пропущена не точка с запятой, а фигурная скобка.

Таким образом, если ваша программа содержит синтаксические ошибки, не стоит бездумно следовать рекомендациям компилятора. Сообщение об ошибке составляется по формальным правилам, вам же нужно выяснять истинные причины ошибок. Желательно проанализировать строки кода программы, предшествующие строке, обозначенной компилятором. Иногда ошибку удается распознать лишь после анализа нескольких строк кода, следующих за той, в которой она была на самом деле обнаружена.

Еще одна простая программа

Вероятно, наиболее распространенной конструкцией в любом языке программирования является оператор присваивания значения переменной. *Переменная* — это именованная ячейка памяти, в которой можно хранить присваиваемое значение. В процессе выполнения программы значение переменной может изменяться. Это означает, что содержимое переменной не фиксировано. В приведенной ниже программе используются две переменные: var1 и var2.

```
/*
   Демонстрация использования переменных.

   Присвойте файлу с исходным кодом имя Example2.java.
*/
class Example2 {
   public static void main(String args[]) {
      int var1; // объявляется переменная ◄——— Объявление переменных
      int var2; // объявляется еще одна переменная

      var1 = 1024; // переменной var1 присваивается значение 1024 ◄——— Присваивание значения переменной

      System.out.println("Переменная var1 содержит " + var1);

      var2 = var1 / 2;

      System.out.print("Переменная var2 содержит var1 / 2: ");
      System.out.println(var2);
   }
}
```

В результате выполнения программы будет получен следующий результат.

```
Переменная var1 содержит 1024
Переменная var2 содержит var1 / 2: 512
```

В этой программе представлен ряд новых понятий. Сначала в следующей строке объявляется переменная var1 целочисленного типа:

```
int var1; // объявляется переменная
```

В Java каждая переменная должна быть объявлена перед ее использованием. При объявлении переменной задается ее тип, т.е. тип данных, которые могут в ней содержаться. В данном случае переменная var1 может содержать целочисленные значения. Для этой цели при объявлении переменной перед ее именем указывается ключевое слово int. Таким образом, в приведенной выше строке кода объявляется переменная var1 типа int.

В следующей строке кода объявляется вторая переменная var2:

```
int var2; // объявляется еще одна переменная
```

Как видите, в этой строке кода используется та же форма объявления переменной, что и в предыдущей. Отличаются они лишь именем переменной.

Общий синтаксис инструкции объявления переменной выглядит следующим образом:

тип имя_переменной;

где *тип* обозначает конкретный тип объявляемой переменной, а *имя_переменной* — ее имя. Помимо int, в Java поддерживаются и другие типы данных.

В следующей строке кода переменной var1 присваивается значение 1024:

```
var1 = 1024; // переменной var1 присваивается значение 1024
```

В Java оператор присваивания обозначается знаком равенства. Он копирует значение, находящееся справа от него, в переменную, указанную слева.

В следующей строке кода значение переменной var1 выводится на экран после символьной строки "Переменная var1 содержит":

```
System.out.println("Переменная var1 содержит " + var1);
```

В этой строке знак + указывает на то, что после символьной строки должно быть выведено значение переменной var1. Этот подход можно обобщить. С помощью оператора + можно объединить несколько элементов в одной инструкции, передав их в качестве параметра методу println().

В следующей строке кода переменной var2 присваивается значение переменной var1, разделенное на 2:

```
var2 = var1 / 2;
```

После выполнения этого кода переменная var2 будет содержать значение 512, тогда как значение переменной var1 останется без изменения. Как и в большинстве других языков программирования, в Java поддерживаются арифметические операции, в том числе перечисленные ниже.

+	Сложение
–	Вычитание
*	Умножение
/	Деление

Рассмотрим следующие две строки кода.

```
System.out.print("Переменная var2 содержит var1 / 2: ");
System.out.println(var2);
```

В первой из них вызывается метод `print()`, выводящий строку "`Переменная var2 содержит var1 / 2:`" *без* последующего символа перевода строки. Это означает, что очередные данные отобразятся в той же строке. Метод `print()` действует аналогично методу `println()`, за исключением того, что его выполнение не завершается переходом на следующую строку. Во второй строке методу `println()` в качестве параметра передается переменная `var2`. С помощью обоих методов, `print()` и `println()`, можно выводить на экран значения любого встроенного в Java типа данных.

Прежде чем переходить к изучению других аспектов Java, необходимо обратить внимание на следующую особенность: в одной строке можно объявить две и более переменных. Для этого достаточно разделить их имена запятыми. Например, переменные `var1` и `var2` можно объявить так:

```
int var1, var2; // обе переменные объявляются в одной строке
```

Другие типы данных

В предыдущем примере программы были использованы переменные типа `int`. Они могут содержать только целые числа и не могут применяться для хранения дробной части. Так, переменная типа `int` может содержать значение `18`, но не `18.3`. Но `int` — это лишь один из нескольких типов данных, определенных в Java. Для хранения чисел, содержащих дробную часть, в Java предусмотрены типы с плавающей точкой, `float` и `double`, которые представляют значения с одинарной и двойной точностью соответственно. Тип `double` используется чаще, чем `float`.

Объявление переменной типа `double` выглядит примерно так:

```
double x;
```

где `x` обозначает имя переменной типа `double`. А поскольку `x` объявлена как переменная с плавающей точкой, в ней можно хранить значения `122.23`, `0.034`, `-19.0` и т.п.

Для того чтобы стали понятнее различия между типами `int` и `double`, рассмотрим следующую программу.

```
/*
   Демонстрация различий между типами int и double.

   Присвойте файлу с исходным кодом имя Example3.java.
*/
class Example3 {
   public static void main(String args[]) {
      int var; // объявление целочисленной переменной
      double x; // объявление переменной с плавающей точкой

      var = 10; // присваивание переменной var значения 10

      x = 10.0; // присваивание переменной x значения 10.0

      System.out.println("Начальное значение переменной var: " + var);
      System.out.println("Начальное значение переменной x: " + x);
      System.out.println(); // печать пустой строки ◄──── Вывод пустой строки

      // Деление значения обеих переменных на 4
      var = var / 4;
      x = x / 4;

      System.out.println("Значение переменной var после деления: " +
                         var);
      System.out.println("Значение переменной x после деления: " + x);
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Начальное значение переменной var: 10
Начальное значение переменной x: 10.0

Значение переменной var после деления: 2 ◄──── Дробная часть числа исчезает
Значение переменной x после деления: 2.5 ◄──── Дробная часть числа сохраняется
```

Как видите, при делении значения переменной `var` на 4 результат получается целым, и программа выводит числовое значение 2, а дробная его часть теряется. В то же время при делении на 4 значения переменной `x` дробная часть ее числового значения сохраняется, и программа отображает правильный результат.

В этой программе показан еще один прием. Для вывода пустой строки на экран достаточно вызвать метод `println()` без параметров.

Спросим у эксперта

ВОПРОС. Почему для хранения целых и дробных значений используются разные типы? Не проще ли создать один универсальный тип для всех числовых значений?

ОТВЕТ. Для написания эффективных программ предусмотрены разные типы данных. Например, целочисленные вычисления выполняются быстрее, чем вычисления с плавающей точкой. Следовательно, если вы не пользуетесь дробными значениями, то лишние накладные расходы, связанные с обработкой содержимого переменной типа `float` или `double`, вам ни к чему. Кроме того, для хранения переменных разных типов требуется разный объем памяти. Предоставляя разработчику возможность выбора типов данных, Java создает благоприятные условия для оптимального использования системных ресурсов. И наконец, для некоторых алгоритмов требуются конкретные типы данных (во всяком случае, они работают с ними более эффективно). Вообще говоря, разные встроенные типы предусмотрены в Java для обеспечения большей гибкости программирования.

Упражнение 1.1 Преобразование галлонов в литры

`GalToLit.java`

Предыдущие примеры программ иллюстрируют некоторые важные особенности языка программирования Java, но не имеют какой-либо практической ценности. Несмотря на то что ваши знания о Java пока еще весьма ограничены, вам вполне по силам написать программу, выполняющую нечто полезное, в частности, преобразование галлонов в литры.

В этой программе объявляются две переменные типа `double`. Одна из них соответствует объему жидкости в галлонах, а вторая — тому же самому объему, но выраженному в литрах. В галлоне содержится 3,7854 литра. Поэтому для преобразования галлонов в литры следует умножить число галлонов на 3,7854. В результате выполнения данной программы объем жидкости отображается как в галлонах, так и в литрах. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл `GalToLit.java`.
2. Сохраните в этом файле следующий код программы.

```
/*
   Упражнение 1.1

   Программа перевода галлонов в литры.

   Присвойте файлу с исходным кодом имя GalToLit.java.
*/
class GalToLit {
   public static void main(String args[]) {
      double gallons; // в этой переменной хранится значение,
                      // выражающее объем жидкости в галлонах
      double liters;  // в этой переменной хранится значение,
                      // выражающее объем жидкости в литрах
```

```
        gallons = 10; // начальное значение соответствует 10 галлонам

        liters = gallons * 3.7854; // перевод в литры

        System.out.println(gallons + " галлонам соответствует " +
                           liters + " литра");
    }
}
```

3. Скомпилируйте программу, введя в командной строке следующую команду:

```
C>javac GalToLit.java
```

4. Запустите программу на выполнение, введя следующую команду:

```
C>java GalToLit
```

   Результат выполнения данной программы выглядит следующим образом:

```
10.0 галлонам соответствует 37.854 литра
```

5. Данная программа преобразует в литры только одно значение объема жидкости, равное десяти галлонам. Изменив значение переменной `gallons`, выражающее объем жидкости в галлонах, вы получите другое результирующее значение, выражающее объем жидкости в литрах.

Две управляющие инструкции

Инструкции в методе выполняются последовательно сверху вниз. Но этот порядок можно изменить, воспользовавшись управляющими инструкциями, предусмотренными в Java. Подробнее об управляющих инструкциях речь пойдет в последующих главах, а пока ограничимся кратким рассмотрением двух таких инструкций. Они будут использованы при написании примеров программ.

Инструкция `if`

Используя условную инструкцию `if`, можно выборочно выполнять отдельные части программы. В Java эта инструкция действует таким же образом, как и инструкция `if` из любого другого языка программирования. Ниже приведена простейшая форма инструкции.

`if(`*`условие`*`)` *`инструкция;`*

Здесь *`условие`* обозначает логическое выражение. Если условие истинно (принимает логическое значение `true`), инструкция выполняется. Если же условие ложно (принимает логическое значение `false`), то инструкция не выполняется. Ниже приведен простой пример применения инструкции `if` в коде.

```
if(10 < 11) System.out.println("10 меньше 11");
```

В данном примере числовое значение `10` меньше `11`, и поэтому условное выражение принимает логическое значение `true`, в результате чего выполняется метод `println()`. Рассмотрим еще один пример с противоположным условием:

```
if(10 < 9) System.out.println("Эта строка не отобразится");
```

В данном случае условие “10 меньше 9” не выполняется, поэтому метод `println()` не вызывается и на экран ничего не выводится.

В Java определен полный набор операторов сравнения, которые могут быть использованы при составлении условных выражений. Эти операторы перечислены ниже.

Оператор	Значение
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Неравно

Обратите внимание на то, что для проверки на равенство следует указать два знака равенства.

Ниже приведен пример программы, демонстрирующий применение инструкции `if`.

```
/*
   Демонстрация применения инструкции if.

   Присвойте файлу с исходным кодом имя IfDemo.java.
*/
class IfDemo {
   public static void main(String args[]) {
      int a, b, c;

      a = 2;
      b = 3;

      if(a < b) System.out.println("a меньше b");

      // Следующая строка никогда не будет выведена
      if(a == b) System.out.println("Вы не должны увидеть этот текст");

      System.out.println();

      c = a - b; // переменная "c" содержит значение -1

      System.out.println("c содержит -1");
      if(c >= 0) System.out.println("c - не отрицательное число");
      if(c < 0) System.out.println("c - отрицательное число");

      System.out.println();

      c = b - a; // переменная "c" теперь содержит значение 1

      System.out.println("c содержит 1");
      if(c >= 0) System.out.println("c - не отрицательное число");
```

```
        if(c < 0) System.out.println("c - отрицательное число");
    }
}
```

Ниже приведен результат выполнения данной программы.

```
a меньше b

c содержит -1
c - отрицательное число

c содержит 1
c - не отрицательное число
```

Обратите внимание еще на одну особенность данной программы. В следующей строке объявляются три переменные, `a`, `b` и `c`, имена которых разделяются запятыми:

```
int a, b, c;
```

Как уже упоминалось ранее, если требуется несколько переменных одного типа, их можно объявить в одной строке. Для этого имена переменных следует разделить запятыми.

Цикл `for`

Циклы позволяют многократно выполнять наборы инструкций. В Java для этой цели предусмотрен целый ряд удобных конструкций циклов. Сначала рассмотрим цикл `for`. Ниже приведена простейшая форма этого цикла.

```
for(инициализация; условие; итерация) инструкция;
```

В самой общей форме исходное значение переменной цикла задается в разделе *инициализация*, а в разделе *условие* содержится логическое выражение, которое, как правило, проверяет значение переменной цикла. Если логическое выражение принимает значение `true`, цикл `for` продолжает выполняться. Если же оно принимает значение `false`, то цикл прекращает выполнение. И наконец, в разделе *итерация* задается порядок изменения переменной цикла на каждой итерации. Ниже приведен пример простой программы, демонстрирующий применение цикла `for`.

```
/*
   Демонстрация применения цикла for.

   Присвойте файлу с исходным кодом имя ForDemo.java.
*/
class ForDemo {
   public static void main(String args[]) {
      int count;

      for(count = 0; count < 5; count = count + 1) ◄── Этот цикл выполняет
         System.out.println("Значение счетчика: " + count);   пять итераций

      System.out.println("Готово!");
   }
}
```

Вот результат выполнения данной программы.

```
Значение счетчика: 0
Значение счетчика: 1
Значение счетчика: 2
Значение счетчика: 3
Значение счетчика: 4
Готово!
```

В данном примере `count` — это переменная цикла. При инициализации ей присваивается значение `0`. В начале каждого шага цикла (включая и первый) проверяется условие `count < 5`. Если это условное выражение принимает логическое значение `true`, вызывается метод `println()`, после чего выполняется итерационная часть цикла. Данная последовательность действий повторяется до тех пор, пока условное выражение не примет логическое значение `false`. В этом случае управление передается следующей после цикла инструкции.

Следует иметь в виду, что в профессионально написанных программах очень редко можно встретить итерационную часть цикла, написанную так, как это сделано в приведенном выше примере программы:

```
count = count + 1;
```

Дело в том, что в Java имеется более эффективный оператор инкремента, который обозначается двумя знаками "плюс" (++) без пробела между ними. В результате выполнения этого оператора значение операнда увеличивается на единицу. Используя оператор инкремента, предыдущее итерационное выражение можно переписать следующим образом:

```
count++;
```

Следовательно, цикл `for` из предыдущего примера программы можно переписать так:

```
for(count = 0; count < 5; count++)
```

Попробуйте внести это изменение в рассматриваемый здесь цикл, и вы увидите, что результаты выполнения программы останутся прежними.

В Java предусмотрен также оператор декремента, обозначаемый двумя знаками "минус" (--) без пробела между ними. При выполнении этого оператора значение операнда уменьшается на единицу.

Создание блоков кода

Еще одним ключевым элементом Java является *блок кода*. Он представляет собой группу инструкций, находящихся между открывающей и закрывающей фигурными скобками. Созданный блок кода становится единым логическим блоком и может использоваться в любом месте программы как одиночная инструкция. В частности, такой блок можно использовать в качестве тела инструкций `if` и `for`. Рассмотрим следующий пример применения блоков в инструкции `if`.

```
if (w < h) {  ←——— Начало блока
   v = w * h;
   w = 0;     ←——— Конец блока
}
```

В данном примере обе инструкции в блоке выполняются в том случае, если значение переменной w меньше значения переменной h. Эти инструкции составляют единый логический блок, и ни одна из них не может быть выполнена без другой. Таким образом, для объединения нескольких выражений в единое логическое целое нужно создать блок кода. Блоки позволяют оформлять многие алгоритмы в удобном для восприятия виде.

Ниже приведен пример программы, в которой блок кода используется для того, чтобы предотвратить деление на нуль.

```
/*
   Демонстрация применения блоков кода.

   Присвойте файлу с исходным кодом имя BlockDemo.java
*/
class BlockDemo {
   public static void main(String args[]) {
      double i, j, d;

      i = 5;
      j = 10;

      // Телом этой инструкции if является целый блок кода
      if(i != 0) {
         System.out.println("i не равно нулю");  ─┐
         d = j / i;                                ├── Телом оператора if
         System.out.print("j / i равно " + d);   ─┘   является весь блок
      }
    }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
i не равно нулю
j / i равно 2.0
```

В данном примере телом инструкции if служит не одна инструкция, а целый блок кода. Если условие, управляющее инструкцией if, принимает логическое значение true, как в данном случае, то выполняются все три инструкции, образующие блок. Попробуйте присвоить переменной i нулевое значение и запустить программу, и вы увидите, что ни одна из инструкций в составе блока не будет выполнена, т.е. весь этот блок будет пропущен.

Спросим у эксперта

ВОПРОС. Приводит ли использование блоков кода к снижению эффективности программы? Иными словами, выполняются ли какие-то дополнительные действия, когда в коде Java встречаются символы { и }?

ОТВЕТ. Нет. Использование блоков кода не влечет за собой никаких дополнительных издержек. Более того, блоки повышают быстродействие, поскольку они упрощают код и позволяют создавать более эффективные алгоритмы. Символы { и } существуют только в исходном коде программы, но не подразумевают выполнение каких-то дополнительных действий.

Как будет показано далее, блоки обладают также другими свойствами и находят иное применение. Но главное их назначение — создание логически неделимых единиц кода.

Использование точки с запятой в коде программы

В Java точка с запятой служит в качестве *разделителя инструкций*. Это означает, что каждая инструкция должна оканчиваться точкой с запятой. Точка с запятой обозначает окончание одной логической единицы кода.

Как вы уже знаете, блок кода — это совокупность логически связанных инструкций, помещаемых между открывающей и закрывающей фигурными скобками. Блок не завершается точкой с запятой. Он представляет собой группу инструкций, и поэтому точка с запятой ставится в конце каждой инструкции, а сам блок завершается лишь закрывающей фигурной скобкой.

В Java конец строки не считается окончанием инструкции. Поэтому не имеет значения, где именно он находится в строке кода. Например, в Java следующие строки кода:

```
x = y;
y = y + 1;
System.out.println(x + " " + y);
```

означают то же самое, что и такая строка кода:

```
x = y; y = y + 1; System.out.println(x + " " + y);
```

Более того, каждый элемент инструкции можно разместить в отдельной строке. Например, следующая запись вполне допустима.

```
System.out.println("Это длинная выходная строка" +
                   x + y + z +
                   "дополнительный вывод");
```

Перенос на новую строку позволяет избегать длинных строк и делать код программы удобочитаемым. Кроме того, разбивая инструкцию на несколько строк, вы предотвращаете автоматический перенос, который зачастую портит внешний вид исходного текста программы.

Стилевое оформление текста программ с помощью отступов

Вы, вероятно, заметили, что некоторые инструкции в предыдущих примерах программ располагались с отступом от начала строки. В Java допускается произвольное оформление исходного кода, а это означает, что расположение инструкций относительно друг друга не имеет особого значения. Но как показывает опыт программирования, располагая некоторые инструкции с отступом от начала строки, можно сделать исходный текст программы более удобным для восприятия. В этой книге широко применяются отступы. Внимательно проанализировав представленные здесь примеры программ, вы, скорее всего, согласитесь, что подобный стиль представления исходного кода вполне оправдан. Так, инструкции, заключенные в фигурные скобки, желательно располагать с отступом. А некоторым инструкциям требуется дополнительный отступ. Подробнее об этом речь пойдет далее.

Упражнение 1.2

Усовершенствованная версия программы для преобразования галлонов в литры

`GalToLitTable.java`

В рассматриваемой здесь усовершенствованной версии программы, переводящей галлоны в литры и созданной в рамках первого упражнения, используются цикл `for`, условная инструкция `if` и блоки кода. В новой версии программы на экран выводится таблица перевода для значений от 1 до 100 (в галлонах). После каждых десяти значений отображается пустая строка. Это достигается благодаря переменной `counter`, которая подсчитывает число выведенных строк. Обратите внимание на особенности применения данной переменной. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл `GalToLitTable.java`.
2. Сохраните в этом файле следующий код программы.

```
/*
   Упражнение 1.2

   Эта программа отображает таблицу перевода галлонов в литры.

   Присвойте файлу с исходным кодом имя GalToLitTable.java.
*/
```

```
class GalToLitTable {
   public static void main(String args[]) {
      double gallons, liters;
      int counter;

      counter = 0;  ◄—— Счетчик строк инициализируется нулевым значением
      for(gallons = 1; gallons <= 100; gallons++) {
         liters = gallons * 3.7854; // преобразование в литры
         System.out.println(gallons + " галлонам соответствует " +
                            liters + " литра.");

         counter++;  ◄—— Увеличивать значение счетчика строк на 1 на каждой итерации цикла
         if(counter == 10) {  ◄—— Если значение счетчика равно 10, вывести пустую строку
            System.out.println();
            counter = 0; // сбросить счетчик строк
         }
      }
   }
}
```

3. Скомпилируйте программу, введя в командной строке следующее:

```
C>javac GalToLitTable.java
```

4. Запустите программу, введя в командной строке такую команду:

```
C>java GalToLitTable
```

5. В результате выполнения данной программы на экран будет выведен следующий результат.

```
1.0 галлонам соответствует 3.7854 литра
2.0 галлонам соответствует 7.5708 литра
3.0 галлонам соответствует 11.356200000000001 литра
4.0 галлонам соответствует 15.1416 литра
5.0 галлонам соответствует 18.927 литра
6.0 галлонам соответствует 22.712400000000002 литра
7.0 галлонам соответствует 26.4978 литра
8.0 галлонам соответствует 30.2832 литра
9.0 галлонам соответствует 34.0686 литра
10.0 галлонам соответствует 37.854 литра

11.0 галлонам соответствует 41.6394 литра
12.0 галлонам соответствует 45.424800000000005 литра
13.0 галлонам соответствует 49.2102 литра
14.0 галлонам соответствует 52.9956 литра
15.0 галлонам соответствует 56.781 литра
16.0 галлонам соответствует 60.5664 литра
17.0 галлонам соответствует 64.3518 литра
18.0 галлонам соответствует 68.1372 литра
19.0 галлонам соответствует 71.9226 литра
20.0 галлонам соответствует 75.708 литра
```

```
21.0 галлонам соответствует 79.49340000000001 литра
22.0 галлонам соответствует 83.2788 литра
23.0 галлонам соответствует 87.0642 литра
24.0 галлонам соответствует 90.84960000000001 литра
25.0 галлонам соответствует 94.635 литра
26.0 галлонам соответствует 98.4204 литра
27.0 галлонам соответствует 102.2058 литра
28.0 галлонам соответствует 105.9912 литра
29.0 галлонам соответствует 109.7766 литра
30.0 галлонам соответствует 113.562 литра
```

Ключевые слова Java

В настоящее время в языке Java определено шестьдесят одно ключевое слово (табл. 1.1). Вместе с синтаксисом операторов и разделителями они образуют определение языка Java. Ключевые слова нельзя использовать в качестве имен переменных, классов или методов. Исключения из этого правила представляют новые контекстно-зависимые ключевые слова, которые появились в JDK 9 в целях поддержки модулей. (Дополнительные сведения по этой теме приведены в главе 15.) Также (начиная с JDK 9) в качестве ключевого слова рассматривается символ подчеркивания: его нельзя применять в качестве имени в программах.

Ключевые слова `const` и `goto` зарезервированы, но не используются. На ранних этапах развития Java для дальнейшего использования были зарезервированы и другие ключевые слова, но в текущем определении (так называемой спецификации) Java определены только ключевые слова, представленные в табл. 1.1.

Таблица 1.1. Ключевые слова Java

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	open	opens	package
private	protected	provides	public	requires	return
short	static	strictfp	super	switch	syncronized
this	throw	throws	to	transient	transitive
try	uses	void	volatile	while	which
_					

Помимо ключевых слов, в Java зарезервированы также логические значения `true`, `false` и `null`. Их нельзя использовать для обозначения переменных, классов и других элементов программ.

Идентификаторы в Java

В Java *идентификатор* обозначает имя метода, переменной или элемента, определяемых пользователем. Идентификатор может содержать один или несколько символов. Имя переменной может начинаться с любой буквы, знака подчеркивания (_) или символа доллара ($). Далее могут следовать буквы, цифры, символы подчеркивания и доллара. Символ подчеркивания обычно применяется для того, чтобы сделать имя более удобным для восприятия, например `line_count`.

В Java символы нижнего и верхнего регистра различаются, т.е. `myvar` и `MyVar` — это имена разных переменных. Ниже приведен ряд идентификаторов, допустимых в Java.

`Test`	`x`	`y2`	`MaxLoad`
`$up`	`_top`	`my_var`	`sample23`

Как упоминалось выше, идентификатор не может начинаться с цифры. Например, идентификатор `12x` недопустим.

В качестве идентификаторов нельзя использовать ключевые слова Java, а также имена стандартных методов, например `println`. Эти ограничения необходимо соблюдать. Кроме того, профессиональный стиль программирования предполагает употребление информативных имен, отражающих назначение соответствующих элементов.

Библиотеки классов Java

В примерах программ, представленных в этой главе, применяются два встроенных в Java метода: `println()` и `print()`. Они являются членами класса `System`, который предопределен в Java и автоматически включается в состав любой программы. В более широком контексте среда Java включает в себя ряд встроенных библиотек классов, содержащих большое количество методов. Они обеспечивают поддержку операций ввода-вывода, операций с символьными строками, сетевого взаимодействия и отображения графики. Стандартные классы также реализуют оконный вывод. Таким образом, Java представляет собой сочетание собственно языка и стандартных классов. Как станет ясно в дальнейшем, многими своими функциональными возможностями язык Java обязан именно библиотекам классов. Поэтому научиться грамотно программировать на Java невозможно, не изучая стандартные классы. На протяжении всей книги

вам будут встречаться описания различных классов и методов из стандартных библиотек. Но в одной книге невозможно описать все библиотеки, поэтому полученные знания основ Java вам придется пополнять в процессе самостоятельной работы.

Вопросы и упражнения для самопроверки

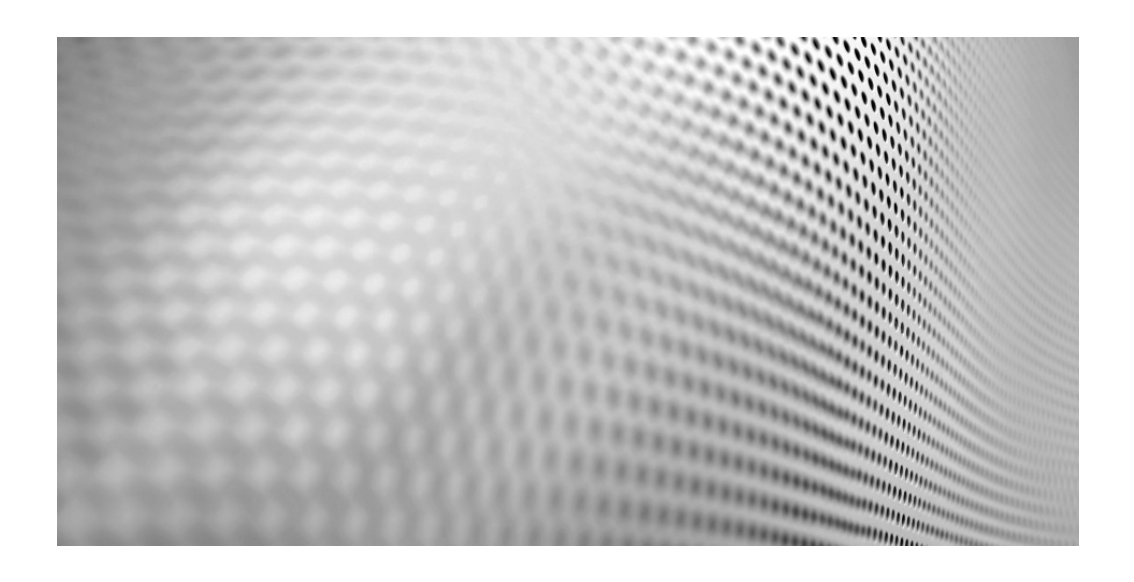
1. Что такое байт-код и почему он так важен для веб-программирования на языке Java?
2. Каковы три ключевых принципа объектно-ориентированного программирования?
3. С чего начинается выполнение программы на Java?
4. Что такое переменная?
5. Какое из перечисленных ниже имен переменных недопустимо?

   А. `count`

   Б. `$count`

   В. `count27`

   Г. `67count`
6. Как создаются однострочные и многострочные комментарии?
7. Как выглядит общая форма условной инструкции `if`? Как выглядит общая форма цикла `for`?
8. Как создать блок кода?
9. Сила тяжести на Луне составляет около 17% земной. Напишите программу, которая вычислила бы ваш вес на Луне.
10. Видоизмените программу, созданную в упражнении 1.2, таким образом, чтобы она выводила таблицу перевода дюймов в метры. Выведите значения длины до 12 футов через каждый дюйм. После каждых 12 дюймов выведите пустую строку. (Один метр приблизительно равен 39,37 дюйма.)
11. Если при вводе кода программы вы допустите опечатку, то какого рода сообщение об ошибке получите?
12. Имеет ли значение, с какой именно позиции в строке начинается инструкция?



Глава 2

Знакомство с типами данных и операторами

В этой главе...

- Примитивные типы данных в Java
- Использование литералов
- Инициализация переменных
- Области действия переменных в методе
- Арифметические операторы
- Операторы сравнения и логические операторы
- Операторы присваивания
- Укороченные операторы присваивания
- Преобразование типов при присваивании
- Приведение несовместимых типов данных
- Преобразование типов в выражениях

Основу любого языка программирования составляют типы данных и операторы, и Java не является исключением из этого правила. Типы данных и операторы определяют область применимости языка и круг задач, которые можно успешно решать с его помощью. В Java поддерживаются самые разные типы данных и операторы, что делает этот язык универсальным и пригодным для написания любых программ.

Эта глава начинается с анализа основных типов данных и наиболее часто используемых операторов. Также будут подробно рассмотрены переменные и выражения.

Почему типы данных столь важны

В связи с тем что Java относится к категории строго типизированных языков программирования, типы данных имеют в нем очень большое значение. В процессе компиляции проверяются типы операндов во всех операторах. И если в программе встречаются недопустимые операции, то ее исходный код не преобразуется в байт-код. Контроль типов позволяет уменьшить количество ошибок и повысить степень надежности программы. В отличие от других языков программирования, где можно не указывать типы данных, хранящихся в переменных, в Java все переменные, выражения и значения строго контролируются на соответствие типов данных. Более того, тип переменной определяет, какие именно операции могут быть выполнены над ней. Операции, разрешенные для одного типа данных, могут оказаться недопустимыми для другого.

Примитивные типы данных Java

Встроенные типы данных в Java разделяются на две категории: объектно-ориентированные и не объектно-ориентированные. Объектно-ориентированные типы данных определяются в классах, о которых речь пойдет в последующих главах. В основу языка Java положено восемь примитивных типов данных, приведенных в табл. 2.1 (их также называют элементарными или простыми). Термин *примитивные* указывает на то, что эти типы данных являются не объектами, а обычными двоичными значениями. Подобные типы данных предусмотрены в языке для того, чтобы повысить эффективность работы программ. Все остальные типы данных Java формируются на основе примитивных типов.

В Java четко определены области действия примитивных типов и диапазон допустимых значений для них. Эти правила должны соблюдаться при создании всех виртуальных машин. А поскольку программы на Java должны быть переносимыми, точное следование этим правилам является одним из основных требований языка. Например, тип `int` остается неизменным в любой исполняющей среде, благодаря чему удается обеспечить реальную переносимость программ. Это означает, что при переходе с одной платформы на другую не придется переписывать код. И хотя строгий контроль типов может привести к незначительному снижению производительности в некоторых исполняющих средах, он является обязательным условием переносимости программ.

Таблица 2.1. Встроенные примитивные типы данных Java

Тип	Описание
`boolean`	Представляет логические значения `true` и `false`
`byte`	8-разрядное целое число
`char`	Символ
`double`	Числовое значение с плавающей точкой двойной точности
`float`	Числовое значение с плавающей точкой одинарной точности
`int`	Целое число
`long`	Длинное целое число
`short`	Короткое число

Целочисленные типы данных

В Java определены четыре целочисленных типа данных: `byte`, `short`, `int` и `long`. Их краткое описание приведено ниже.

Тип	Разрядность, бит	Диапазон допустимых значений
`byte`	8	от –128 до 127
`short`	16	от –32768 до 32767

Окончание табл. 2.1

Тип	Разрядность, бит	Диапазон допустимых значений
`int`	32	от –2147483648 до 2147483647
`long`	64	от –9223372036854775808 до 9223372036854775807

Как следует из приведенной выше таблицы, целочисленные типы данных предполагают как положительные, так и отрицательные значения. В Java не поддерживаются целочисленные значения без знака, т.е. только положительные целые числа. Во многих других языках программирования широко применяются целочисленные типы данных без знака, но разработчики Java посчитали их применение излишним.

Примечание

В исполняющей среде Java для хранения простых типов может формально выделяться любой объем памяти, но диапазон допустимых значений остается неизменным.

Из всех целочисленных типов данных чаще всего применяется `int`. Переменные типа `int` нередко используются в качестве переменных циклов, индексов массивов и, конечно же, для выполнения универсальных операций над целыми числами.

Если диапазон значений, допустимых для типа `int`, вас не устраивает, можно выбрать тип `long`. Ниже приведен пример программы для расчета числа кубических дюймов в кубе, длина, ширина и высота которого равны одной миле.

```
/*
   Расчет числа кубических дюймов в кубе объемом в 1 куб. милю
*/
class Inches {
   public static void main(String args[]) {
      long ci;
      long im;

      im = 5280 * 12;

      ci = im * im * im;

      System.out.println("В одной кубической миле содержится " +
                         ci + " кубических дюймов");
   }
}
```

Результат выполнения данной программы выглядит следующим образом:

```
В одной кубической миле содержится 254358061056000 кубических дюймов
```

Очевидно, что результирующее значение не умещается в переменной типа `int`.

Наименьшим диапазоном допустимых значений среди всех целочисленных типов обладает тип `byte`. Переменные типа `byte` очень удобны для обработки исходных двоичных данных, которые могут оказаться несовместимыми с другими встроенными в Java типами данных. Тип `short` предназначен для хранения небольших целых чисел. Переменные данного типа пригодны для хранения значений, изменяющихся в относительно небольших пределах по сравнению со значениями типа `int`.

Спросим у эксперта

ВОПРОС. Как упоминалось выше, существуют четыре целочисленных типа данных: `int`, `short`, `long` и `byte`. Но говорят, что тип `char` в Java также считается целочисленным. В чем здесь дело?

ОТВЕТ. Формально в спецификации Java определена категория целочисленных типов данных, в которую входят типы `byte`, `short`, `int`, `long` и `char`. Такие типы называют целочисленными, поскольку они могут хранить целые двоичные значения. Первые четыре типа предназначены для представления целых чисел, а тип `char` — для представления символов. Таким образом, тип `char` принципиально отличается от остальных четырех целых типов данных. Учитывая это отличие, тип `char` рассматривается в данной книге отдельно.

Типы данных с плавающей точкой

Как говорилось в главе 1, типы с плавающей точкой позволяют хранить числовые значения с дробной частью. Существуют два типа данных с плавающей точкой: `float` и `double`. Они представляют числовые значения с одинарной и двойной точностью соответственно. Разрядность данных типа `float` составляет 32 бита, а данных типа `double` — 64 бита.

Тип `double` применяется намного чаще, чем `float`, поскольку во всех математических функциях из библиотек классов Java используются значения типа `double`. Например, метод `sqrt()`, определенный в стандартном классе `Math`, возвращает значение `double`, являющееся квадратным корнем значения аргумента этого метода, также представленного типом `double`. Ниже приведен фрагмент кода, в котором метод `sqrt()` используется для расчета длины гипотенузы при условии, что заданы длины катетов.

```
/*
   Определение длины гипотенузы исходя из длины катетов,
   по теореме Пифагора
*/
class Hypot {
   public static void main(String args[]) {
      double x, y, z;
```

```
        x = 3;
        y = 4;

        z = Math.sqrt(x*x + y*y);

        System.out.println("Длина гипотенузы: " +z);
    }
}
```

Обратите внимание на вызов метода sqrt(). Перед именем метода указывается имя класса, членом которого он является.

В результате выполнения этого фрагмента кода будет получен следующий результат:

```
Длина гипотенузы: 5.0
```

Как упоминалось выше, метод sqrt() определен в стандартном классе Math. Обратите внимание на вызов этого метода в приведенном выше фрагменте кода: перед его именем указывается имя класса. Аналогичным образом перед именем метода println() указываются имена классов System.out. Имя класса указывается не перед всеми стандартными методами, но для некоторых из них целесообразно применять именно такой способ.

Символы

В отличие от других языков, в Java символы не являются 8-битовыми значениями. Вместо этого в Java используется кодировка Unicode, позволяющая представлять символы всех письменных языков. В Java тип char хранит 16-разрядное значение без знака в диапазоне от 0 до 65536. Стандартный набор 8-разрядных символов кодировки ASCII является подмножеством стандарта Unicode. В нем коды символов находятся в пределах от 0 до 127. Следовательно, символы в коде ASCII по-прежнему допустимы в Java.

Переменной символьного типа может быть присвоено значение, которое записывается в виде символа, заключенного в одинарные кавычки. В приведенном ниже фрагменте кода показано, каким образом переменной ch присваивается буква X.

```
char ch;
ch = 'X';
```

Отобразить значение типа char можно с помощью метода println(). В приведенной ниже строке кода показано, каким образом этот метод вызывается для вывода на экран значения символа, хранящегося в переменной ch:

```
System.out.println("Это символ " + ch);
```

Поскольку тип char представляет 16-разрядное значение без знака, над переменной символьного типа можно выполнять различные арифметические операции. Рассмотрим в качестве примера следующую программу.

```
// С символьными переменными можно обращаться
// как с целочисленными
class CharArithDemo {
   public static void main(String args[]) {
      char ch;

      ch = 'X';
      System.out.println("ch содержит " + ch);

      ch++; // инкрементировать переменную ch ◄——— Переменную типа char
      System.out.println("теперь ch содержит " + ch);   можно инкрементировать

      ch = 90; // присвоить переменной ch значение 'Z' ◄——— Переменной
      System.out.println("теперь ch содержит " + ch);       типа char
   }                                                        можно присвоить
}                                                           целочисленное
                                                            значение
```

Ниже приведен результат выполнения данной программы.

```
ch содержит X
теперь ch содержит Y
теперь ch содержит Z
```

В приведенной выше программе переменной ch сначала присваивается значение кода буквы 'X'. Затем содержимое ch увеличивается на единицу, в результате чего оно превращается в код буквы 'Y' — следующего по порядку символа в наборе ASCII (а также в наборе Unicode). После этого переменной ch присваивается значение 90, представляющее букву 'Z' в наборе символов ASCII (и в Unicode). А поскольку символам набора ASCII соответствуют первые 127 значений набора Unicode, то все приемы, обычно применяемые для манипулирования символами в других языках программирования, будут работать и в Java.

Спросим у эксперта

ВОПРОС. Почему в Java для кодировки символов применяется стандарт Unicode?

ОТВЕТ. Язык Java изначально разрабатывался для международного применения. Поэтому возникла необходимость в наборе символов, способном представлять все письменные языки. Именно для этой цели и был разработан стандарт Unicode. Очевидно, что применение этого стандарта для таких языков, как английский, немецкий, испанский и французский, сопряжено с дополнительными издержками, поскольку для символа, который вполне может быть представлен восемью битами, приходится выделять 16 бит. Это та цена, которую приходится платить за обеспечение переносимости программ.

Логический тип данных

Тип boolean представляет логические значения "истина" и "ложь", для которых в Java зарезервированы ключевые слова true и false соответственно. Следовательно, переменная или выражение типа boolean может принимать одно из этих двух значений.

Ниже приведен пример программы, демонстрирующий применение типа данных boolean в коде.

```
// Демонстрация использования логических значений
class BoolDemo {
   public static void main(String args[]) {
      boolean b;

      b = false;
      System.out.println("Значение b: " + b);
      b = true;
      System.out.println("Значение b: " + b);

      // Логическое значение можно использовать для
      // управления условной инструкцией if
      if(b) System.out.println("Эта инструкция выполняется");

      b = false;
      if(b) System.out.println("Эта инструкция не выполняется");

      // В результате сравнения получается логическое значение
      System.out.println("Результат сравнения 10 > 9: " + (10 > 9));
   }
}
```

Результат выполнения данной программы будет таким.

```
Значение b: false
Значение b: true
Эта инструкция выполняется
Результат сравнения 10 > 9: true
```

Анализируя эту программу, необходимо отметить следующее. Во-первых, метод println(), обрабатывая логическое значение, отображает символьные строки true и false. Во-вторых, значение логической переменной может быть само по себе использовано для управления условной инструкцией if. Это означает, что в следующем выражении нет необходимости:

```
if(b == true) ...
```

И в-третьих, результатом выполнения операции сравнения, например <, является логическое значение. Именно поэтому при передаче методу println() выражения (10 > 9) отображается логическое значение true. Скобки в данном случае необходимы, потому что оператор + имеет более высокий приоритет по сравнению с оператором >.

Упражнение 2.1

Вычисление расстояния до места вспышки молнии

`Sound.java`

В данном упражнении нам предстоит написать программу, вычисляющую расстояние (в футах) до источника звука, возникающего из-за грозового разряда. Звук распространяется в воздухе со скоростью, приблизительно равной 1100 фут/с (330 м/с). Следовательно, зная промежуток времени между теми моментами, когда наблюдатель увидит вспышку молнии и услышит сопровождающий ее раскат грома, можно рассчитать расстояние до нее. Допустим, что этот промежуток времени составляет 7,2 секунды. Ниже приведено поэтапное описание процесса создания программы.

1. Создайте новый файл `Sound.java`.
2. Для расчета искомого расстояния потребуются числовые значения с плавающей точкой. Почему? Потому что упомянутое выше числовое значение промежутка времени содержит дробную часть. И хотя для расчета достаточно точности, обеспечиваемой типом `float`, в данном примере будет использован тип `double`.
3. Для вычисления искомого расстояния умножьте 1100 на 7,2, а полученный результат присвойте переменной типа `double`.
4. Выведите результат вычислений на экран.

   Ниже приведен исходный код программы, находящейся в файле `Sound.java`.

```
/*
   Упражнение 2.1

   Вычисление расстояние до места вспышки молнии, звук от которого
   доходит до наблюдателя через 7,2 секунды.
*/
class Sound {
   public static void main(String args[]) {
      double dist;

      dist = 1100 * 7.2 ;

      System.out.println("Расстояние до места вспышки молнии " +
                         "составляет " + dist + " футов");
   }
}
```

5. Скомпилируйте программу и запустите ее на выполнение. Вы получите следующий результат:

```
Расстояние до места вспышки молнии составляет 7920.0 футов
```

6. А теперь усложним задачу. Рассчитать расстояние до крупного объекта, например скалы, можно по времени прихода эхо. Так, если вы хлопнете в ладоши, время, через которое вернется эхо, будет равно времени прохождения звука в прямом и обратном направлении. Разделив этот промежуток времени на два, вы получите время прохождения звука от вас до объекта. Полученное значение можно затем использовать для расчета расстояния до объекта. Видоизмените рассмотренную выше программу, используя при вычислении значение промежутка времени до прихода эха.

Литералы

В Java *литералы* применяются для представления постоянных значений в форме, удобной для восприятия. Например, число `100` является литералом. Литералы часто называют *константами*. Как правило, структура литералов и их использование интуитивно понятны. Они уже встречались в рассмотренных ранее примерах программ, а теперь пришло время дать им формальное определение.

В Java предусмотрены литералы для всех простых типов. Способ представления литерала зависит от типа данных. Как пояснялось ранее, константы, соответствующие символам, заключаются в одинарные кавычки. Например, `'a'` и `'%'` являются символьными константами.

Целочисленные константы записываются как числа без дробной части. Например, целочисленными константами являются `10` и `100`. При создании константы с плавающей точкой необходимо указывать десятичную точку, после которой следует дробная часть. Например, `11.123` — это константа с плавающей точкой. В Java поддерживается и так называемый экспоненциальный формат представления чисел с плавающей точкой.

По умолчанию целочисленные литералы относятся к типу `int`. Если же требуется определить литерал типа `long`, после числа следует указать букву `l` или `L`. Например, `12` — это константа типа `int`, а `12L` — константа типа `long`.

По умолчанию литералы с плавающей точкой относятся к типу `double`. А для того чтобы задать литерал типа `float`, следует указать после числа букву `f` или `F`. Так, например, к типу `float` относится литерал `10.19F`.

Несмотря на то что целочисленные литералы по умолчанию создаются как значения типа `int`, их можно присваивать переменным типа `char`, `byte`, `short` и `long`. Присваиваемое значение приводится к целевому типу. Переменной типа `long` можно также присвоить любое значение, представленное целочисленным литералом.

В версии JDK 7 появилась возможность включать в литералы (как целочисленные, так и с плавающей точкой) знак подчеркивания. Благодаря этому упрощается восприятие числовых значений, состоящих из нескольких цифр. А при

компиляции знаки подчеркивания просто удаляются из литерала. Ниже приведен пример литерала со знаком подчеркивания.

```
123_45_1234
```

Этот литерал задает числовое значение 123451234. Пользоваться знаками подчеркивания особенно удобно при кодировании номеров деталей, идентификаторов заказчиков и кодов состояния, которые обычно состоят из целых групп цифр.

Шестнадцатеричные, восьмеричные и двоичные литералы

Вам, вероятно, известно, что при написании программ бывает удобно пользоваться числами, представленными в системе счисления, отличающейся от десятичной. Для этой цели чаще всего выбираются восьмеричная (с основанием 8) и шестнадцатеричная (с основанием 16) системы счисления. В *восьмеричной* системе используются цифры от 0 до 7, а число 10 соответствует числу 8 в десятичной системе. В *шестнадцатеричной* системе используются цифры от 0 до 9, а также буквы от A до F, которыми обозначаются числа 10, 11, 12, 13, 14 и 15 в десятичной системе, тогда как число 10 в шестнадцатеричной системе соответствует десятичному числу 16. Восьмеричная и шестнадцатеричная системы используются очень часто в программировании, и поэтому в языке Java предусмотрена возможность представления целочисленных констант (или литералов) в восьмеричной и шестнадцатеричной форме. Шестнадцатеричная константа должна начинаться с символов `0x` или `0X` (цифра 0, после которой следует буква 'x' или 'X'). А восьмеричная константа начинается с нуля. Ниже приведены примеры таких констант.

```
hex = 0xFF; // соответствует десятичному числу 255
oct = 011; // соответствует десятичному числу 9
```

Любопытно, что в Java допускается также задавать шестнадцатеричные литералы в формате с плавающей точкой, хотя они применяются очень редко.

В версии JDK 7 появилась также возможность задавать целочисленный литерал в двоичной форме. Для этого перед целым числом достаточно указать символы `0b` или `0B`. Например, следующий литерал определяет целое значение 12 в двоичной форме:

```
0b1100
```

Управляющие последовательности символов

Заключение символьных констант в одинарные кавычки подходит для большинства печатных символов, но некоторые непечатаемые символы, например символ возврата каретки, становятся источником проблем при работе с текстовыми редакторами. Кроме того, некоторые знаки, например одинарные и двойные кавычки, имеют специальное назначение, и потому их нельзя непосредственно указывать в качестве литерала. По этой причине в языке Java

предусмотрены специальные *управляющие последовательности*, начинающиеся с обратной косой черты (помещение обратной косой черты перед символом называют экранированием символа). Эти последовательности перечислены в табл. 2.2. Они используются в литералах вместо непечатаемых символов, которые они представляют.

Таблица 2.2. Управляющие последовательности символов

Управляющая последовательность	Описание
`\'`	Одинарная кавычка
`\"`	Двойная кавычка
`\\`	Обратная косая черта
`\r`	Возврат каретки
`\n`	Перевод строки
`\f`	Перевод страницы
`\t`	Горизонтальная табуляция
`\b`	Возврат на одну позицию
`\`*ddd*	Восьмеричная константа (где *ddd* — восьмеричное число)
`\u`*xxxx*	Шестнадцатеричная константа (где *xxxx* — шестнадцатеричное число)

Ниже приведен пример присваивания переменной ch символа табуляции.

```
ch = '\t';
```

А в следующем примере переменной ch присваивается одинарная кавычка:

```
ch = '\'';
```

Строковые литералы

В Java предусмотрены также литералы для представления символьных строк. *Символьная строка* — это набор символов, заключенный в двойные кавычки:

```
"Это тест"
```

Примеры строковых литералов не раз встречались в рассмотренных ранее примерах программ. В частности, они передавались в качестве аргументов методу `println()`.

Помимо обычных символов, строковый литерал также может содержать вышеупомянутые управляющие последовательности. Рассмотрим в качестве примера следующую программу, в которой применяются управляющие последовательности `\n` и `\t`.

```
// Демонстрация управляющих последовательностей в
// символьных строках
class StrDemo {
```

```
    public static void main(String args[]) {
        System.out.println("Первая строка\nВторая строка");
        System.out.println("A\tB\tC");
        System.out.println("D\tE\tF");
    }
}
```

Используйте последовательность \n для вставки символа перевода строки

Используйте табуляцию для выравнивания вывода

Ниже приведен результат выполнения данной программы.

```
Первая строка
Вторая строка
A       B       C
D       E       F
```

Спросим у эксперта

ВОПРОС. Представляет ли строка, состоящая из одного символа, то же самое, что и символьный литерал? Например, есть ли разница между "k" и 'k'?

ОТВЕТ. Это разные вещи. Строки следует отличать от символов. Символьный литерал представляет один символ и относится к типу char. А строка, содержащая даже один символ, все равно остается строкой. Несмотря на то что строки состоят из символов, они относятся к разным типам данных.

Обратите внимание на использование управляющей последовательности \n, применяемой для перевода строки в приведенном выше примере программы. Для вывода на экран нескольких символьных строк вовсе не обязательно вызывать метод println() несколько раз подряд. Достаточно ввести в строку символы **\n**, и при выводе в этом месте произойдет переход на новую строку.

Подробнее о переменных

О переменных уже шла речь в главе 1, а здесь они будут рассмотрены более подробно. Как вы уже знаете, переменная объявляется так:

тип имя_переменной;

где *тип* обозначает конкретный тип объявляемой переменной, а *имя_переменной* — ее имя. Объявить можно переменную любого допустимого типа, включая рассмотренные ранее простые типы. Когда объявляется переменная, создается экземпляр соответствующего типа. Следовательно, возможности переменной определяются ее типом. Например, переменную типа boolean нельзя использовать для хранения значения с плавающей точкой. На протяжении всего времени жизни переменной ее тип остается неизменным. Так, переменная int не может превратиться в переменную char.

В Java каждая переменная должна быть объявлена перед ее использованием. Ведь компилятору необходимо знать, данные какого типа содержит переменная,

и лишь тогда он сможет правильно скомпилировать инструкцию, в которой используется переменная. Объявление переменных позволяет также осуществлять строгий контроль типов в Java.

Инициализация переменных

Прежде чем использовать переменную в выражении, ей нужно присвоить значение. Сделать это можно, в частности, с помощью уже знакомого вам оператора присваивания. Существует и другой способ: инициализировать переменную при ее объявлении. Для этого достаточно указать после имени переменной знак равенства и требуемое значение. Ниже приведена общая форма инициализации переменной:

```
тип переменная = значение;
```

где *значение* обозначает конкретное значение, которое получает *переменная* при ее создании, причем тип значения должен соответствовать указанному типу переменной. Ниже приведен ряд примеров инициализации переменных.

```
int count = 10; // присваивание переменной count начального
                // значения 10
char ch = 'S';  // инициализация переменной ch буквой S
float f = 1.2F; // инициализация переменной f
                // числовым значением 1.2
```

Присваивать начальные значения переменным можно и в том случае, если в одной строке объявляется несколько переменных:

```
int a, b = 8, c = 19, d; // инициализация переменных b и c
```

В данном случае инициализируются переменные b и c.

Динамическая инициализация

В приведенных выше примерах в качестве значений, присваиваемых переменным, использовались только константы. Но в Java поддерживается также динамическая инициализация, при которой можно использовать любые выражения, допустимые в момент объявления переменной. Ниже приведен пример простой программы, в которой объем цилиндра рассчитывается на основании значений его радиуса и высоты.

```
//  Демонстрация динамической инициализации
class DynInit {
   public static void main(String args[]) {
      double radius = 4, height = 5;

      // Переменная volume инициализируется динамически
      // во время выполнения программы
      double volume = 3.1416 * radius * radius * height;

      System.out.println("Объем: " + volume);
   }
}
```

Переменная volume динамически инициализируется во время выполнения

В данном примере используются три локальные переменные: `radius`, `height` и `volume`. Первые две из них инициализируются константами, а для присвоения значения переменной `volume` применяется динамическая инициализация, в ходе которой вычисляется объем цилиндра. В выражении динамической инициализации можно использовать любой определенный к этому моменту элемент, в том числе вызовы методов, другие переменные и литералы.

Область действия и время жизни переменных

Все использовавшиеся до сих пор переменные объявлялись в начале метода `main()`. Но в Java можно объявлять переменные в любом блоке кода. Как пояснялось в главе 1, блок начинается с открывающей фигурной скобки и оканчивается закрывающей фигурной скобкой. Блок определяет *область действия (видимости) переменных*. Начиная новый блок, вы всякий раз создаете новую область действия. По сути, область действия определяет доступность объектов из других частей программы и время их жизни.

Во многих языках программирования поддерживаются две общие категории областей действия: глобальная и локальная. И хотя они поддерживаются и в Java, тем не менее не являются наиболее подходящими понятиями для категоризации областей действия объектов. Намного большее значение в Java имеют области, определяемые классом и методом. Об областях действия, определяемых классом (и объявляемых в них переменных), речь пойдет позже, когда дойдет черед до рассмотрения классов. А пока исследуем только те области действия, которые определяются методами или внутри методов.

Начало области действия, определяемой методом, обозначается открывающей фигурной скобкой. Если для метода предусмотрены параметры, то они также входят в область его действия.

Как правило, переменные, объявленные в некоторой области действия, невидимы (и, следовательно, недоступны) за ее пределами. Таким образом, объявляя переменную в некоторой области действия, вы тем самым ограничиваете пределы ее действия и защищаете ее от нежелательного доступа и видоизменения. На самом деле правила определения области действия служат основанием для инкапсуляции.

Области действия могут быть вложенными. Открывая новый блок кода, вы создаете новую, вложенную область действия. Такая область заключена во внешней области. Это означает, что объекты, объявленные во внешней области действия, будут доступны для кода во внутренней области, но не наоборот. Объекты, объявленные во внутренней области действия, недоступны во внешней области.

Для того чтобы лучше понять принцип действия вложенных областей действия, рассмотрим следующий пример программы.

```
// Демонстрация области действия блока кода
class ScopeDemo {
   public static void main(String args[]) {
      int x; // Эта переменная доступна для всего кода в
             // методе main

      x = 10;
      if(x == 10) { // Начало новой области действия

         int y = 20; // Эта переменная доступна только
                     // в данном блоке

         // Обе переменные, "x" и "y", доступны в данном блоке кода

         System.out.println("x и y: " + x + " " + y);
         x = y * 2;
      }

      // y = 100; // Ошибка! В этом месте переменная "y" недоступна

      // А переменная "x" по-прежнему доступна
      System.out.println("x - это " + x);
   }
}
```

Здесь переменная y находится вне своей области действия

Как следует из комментариев к приведенной выше программе, переменная x определяется в начале области действия метода `main()` и доступна для всего кода, содержащегося в этом методе. В блоке условной инструкции `if` объявляется переменная y. Этот блок определяет область действия переменной y, и, следовательно, она доступна только в нем. Именно поэтому закомментирована строка кода `y = 100;`, находящаяся за пределами данного блока. Если удалить символы комментариев, то при компиляции программы появится сообщение об ошибке, поскольку переменная y недоступна для кода за пределами блока, в котором она объявлена. В то же время в блоке условной инструкции `if` можно пользоваться переменной x, потому что код в блоке, который определяет вложенную область действия, имеет доступ к переменным из внешней, охватывающей его области действия.

Переменные можно объявлять в любом месте блока кода, но сделать это следует перед тем, как пользоваться ими. Именно поэтому переменная, определенная в начале метода, доступна для всего кода этого метода. А если объявить переменную в конце блока, то такое объявление окажется бесполезным, поскольку переменная станет вообще недоступной для кода.

Следует также иметь в виду, что переменные, созданные в области их действия, удаляются, как только управление в программе передается за пределы этой области. Следовательно, после выхода из области действия переменной содержащееся в ней значение теряется. В частности, переменные, объявленные

в теле метода, не хранят значения в промежутках между последовательными вызовами этого метода. Таким образом, время жизни переменной ограничивается областью ее действия.

Если при объявлении переменной осуществляется ее инициализация, то переменная будет повторно инициализироваться при каждом входе в тот блок, в котором она объявлена. Рассмотрим в качестве примера следующую программу.

```
// Демонстрация времени жизни переменной
class VarInitDemo {
   public static void main(String args[]) {
      int x;

      for(x = 0; x < 3; x++) {
         int y = -1; // переменная y инициализируется
                     // при каждом входе в блок
         System.out.println("y: " + y); // всегда выводится
                                        // значение -1
         y = 100;
         System.out.println("Измененное значение y: " + y);
      }
   }
}
```

Ниже приведен результат выполнения данной программы.

```
y: -1
Измененное значение y: 100
y: -1
Измененное значение y: 100
y: -1
Измененное значение y: 100
```

Как видите, на каждом шаге цикла `for` переменная `y` инициализируется значением `-1`. Затем ей присваивается значение `100`, но по завершении блока кода данного цикла оно теряется.

Для правил области действия в Java характерна следующая особенность: во вложенном блоке нельзя объявлять переменную, имя которой совпадает с именем переменной во внешнем блоке. Рассмотрим пример программы, в которой предпринимается попытка объявить две переменные с одним и тем же именем в разных областях действия, и поэтому такая программа не пройдет компиляцию.

```
/*
   В этой программе предпринимается попытка объявить во
   внутренней области действия переменную с таким же именем,
   как и у переменной, объявленной во внешней области действия.

   *** Эта программа не пройдет компиляцию ***
*/
```

```
class NestVar {
   public static void main(String args[]) {
      int count; ◄──────────────────────────────────────────┐
                                                             │
      for(count = 0; count < 10; count = count+1) {         │
         System.out.println("Значение count: " + count);     │
                                                          Нельзя объявлять переменную
         int count; // Недопустимо!!! ◄───────────────── count, поскольку ранее она
         for(count = 0; count < 2; count++)               уже была объявлена
            System.out.println("В этой программе есть ошибка!");
      }
   }
}
```

Если у вас имеется определенный опыт программирования на С или С++, то вам известно, что в этих языках отсутствуют какие-либо ограничения на имена переменных, объявляемых во внутренней области действия. Так, в С и С++ объявление переменной `count` в блоке внешнего цикла `for` из приведенного выше примера программы вполне допустимо, несмотря на то что такая же переменная уже объявлена во внешнем блоке. В этом случае переменная во внутреннем блоке скрывает переменную из внешнего блока. Создатели Java решили, что подобное сокрытие имен переменных может привести к программным ошибкам, и поэтому запретили его.

Операторы

Язык Java предоставляет множество *операторов*, позволяющих выполнять определенные действия над исходными значениями, называемыми *операндами*, для получения результирующего значения. Большинство операторов может быть отнесено к одной из следующих четырех категорий: арифметические, побитовые, логические и сравнения. Кроме того, в Java предусмотрены другие операторы, имеющие специальное назначение. В этой главе будут рассмотрены арифметические и логические операторы, а также операторы сравнения и присваивания. О побитовых операторах и дополнительных операциях речь пойдет позже.

Арифметические операторы

В Java определены следующие арифметические операторы.

Оператор	Выполняемое действие
+	Сложение (а также унарный плюс)
–	Вычитание (а также унарный минус)
*	Умножение
/	Деление

Окончание таблицы

Оператор	Выполняемое действие
%	Деление по модулю (остаток от деления)
++	Инкремент
--	Декремент

Операторы +, -, * и / имеют в Java тот же смысл, что и в любом другом языке программирования в частности и в математике вообще, т.е. выполняют обычные арифметические действия. Их можно применять к любым числовым данным встроенных типов, а также к объектам типа char.

Несмотря на то что арифметические операторы общеизвестны, у них имеются некоторые особенности, требующие специального пояснения. Во-первых, если оператор / применяется к целым числам, остаток от деления игнорируется. Например, результат целочисленного деления 10 / 3 равен 3. Для получения остатка от деления используется оператор деления по модулю %. В Java эта операция выполняется так же, как и в других языках программирования. Например, результатом вычисления выражения 10 % 3 будет 1. Оператор % применим не только к целым числам, но и к числам с плавающей точкой. Следовательно, в результате вычисления выражения 10.0 % 3.0 также будет получено значение 1. Ниже приведен пример программы, демонстрирующий использование оператора %.

```
// Демонстрация использования оператора %
class ModDemo {
   public static void main(String args[]) {
      int iresult, irem;
      double dresult, drem;

      iresult = 10 / 3;
      irem = 10 % 3;

      dresult = 10.0 / 3.0;
      drem = 10.0 % 3.0;

      System.out.println("Результат и остаток от деления 10 / 3: " +
                         iresult + " " + irem);
      System.out.println("Результат и остаток от деления
                         10.0 / 3.0: " + dresult + " " + drem);
   }
}
```

В результате выполнения этой программы будут получены следующие результаты.

```
Результат и остаток от деления 10 / 3: 3 1
Результат и остаток от деления 10.0 / 3.0: 3.3333333333333335 1.0
```

Как видите, оператор % дает остаток от деления как целых чисел, так и чисел с плавающей точкой.

Инкремент и декремент

Операторы ++ и --, выполняющие положительное (инкремент) и отрицательное (декремент) приращение значений, уже были рассмотрены в главе 1. Как будет показано ниже, эти операторы имеют ряд интересных особенностей. Рассмотрим подробнее, что происходит при выполнении операций инкремента и декремента.

Оператор инкремента добавляет к своему операнду единицу, а оператор декремента вычитает единицу из операнда. Следовательно, операция

```
x = x + 1;
```

дает тот же результат, что и операция

```
x++;
```

а операция

```
x = x - 1;
```

дает тот же результат, что и операция

```
x--;
```

Операторы инкремента и декремента могут записываться в одной из двух форм: префиксной (знак операции предшествует операнду) и постфиксной (знак операции следует за операндом). Например, оператор

```
x = x + 1;
```

можно записать так:

```
++x; // префиксная форма
```

или так:

```
x++; // постфиксная форма
```

В приведенных выше примерах результат не зависит от того, какая из форм записи применена. Но при вычислении более сложных выражений применение этих форм будет давать различные результаты. Общее правило таково: префиксной форме записи операций инкремента и декремента соответствует изменение значения операнда до его использования в соответствующем выражении, а постфиксной — после его использования. Рассмотрим конкретный пример.

```
x = 10;
y = ++x;
```

В результате выполнения соответствующих действий значение переменной y будет равно 11. Но если изменить код так, как показано ниже, то результат будет другим.

```
x = 10;
y = x++;
```

Теперь значение переменной y равно 10. При этом в обоих случаях значение переменной x будет равно 11. Возможность контролировать момент выполнения операции инкремента или декремента дает немало преимуществ в процессе написании программ.

Операторы сравнения и логические операторы

Операторы сравнения отличаются от *логических операторов* тем, что первые определяют отношения между значениями, а вторые связывают между собой логические значения (`true` или `false`), получаемые в результате определения отношений между значениями. Операторы сравнения возвращают логическое значение `true` или `false` и поэтому нередко используются совместно с логическими операторами. По этой причине они и рассматриваются вместе.

Ниже перечислены операторы сравнения.

Оператор	Значение
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Ниже перечислены логические операторы.

Оператор	Значение
&	И
|	ИЛИ
^	Исключающее ИЛИ
||	Укороченное ИЛИ
&&	Укороченное И
!	НЕ

Результатом выполнения операции сравнения или логической операции является логическое значение типа `boolean`.

В Java все объекты могут быть проверены на равенство или неравенство с помощью операторов == и != соответственно. В то же время операторы <, >, <= и >= могут применяться только к тем типам данных, для которых определено отношение порядка. Следовательно, к данным числовых типов и типа `char` можно применять все операции сравнения. Логические же значения типа `boolean` можно проверять только на равенство или неравенство, поскольку истинные (`true`) и ложные (`false`) значения не имеют отношения порядка. Например, выражение `true > false` не имеет смысла в Java.

Операнды логических операторов должны иметь тип `boolean`, как, впрочем, и результаты выполнения этих операций. Встроенным операторам Java `&`, `|`, `^`

и ! соответствуют логические операции **И**, **ИЛИ**, исключающее **ИЛИ** и **НЕ**. Результаты их применения к логическим операндам p и q представлены в следующей таблице.

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

Отсюда видно, что результат выполнения логической операции "исключающее **ИЛИ**" будет истинным (true) в том случае, если один и только один из ее операндов имеет логическое значение true.

Приведенный ниже пример программы демонстрирует применение некоторых операторов сравнения и логических операторов.

```
// Демонстрация использования операторов сравнения
// и логических операторов
class RelLogOps {
   public static void main(String args[]) {
      int i, j;
      boolean b1, b2;

      i = 10;
      j = 11;
      if(i < j) System.out.println("i < j");
      if(i <= j) System.out.println("i <= j");
      if(i != j) System.out.println("i != j");
      if(i == j) System.out.println("Это не будет выполнено");
      if(i >= j) System.out.println("Это не будет выполнено");
      if(i > j) System.out.println("Это не будет выполнено");

      b1 = true;
      b2 = false;
      if(b1 & b2) System.out.println("Это не будет выполнено");
      if(!(b1 & b2)) System.out.println("!(b1 & b2): true");
      if(b1 | b2) System.out.println("b1 | b2: true");
      if(b1 ^ b2) System.out.println("b1 ^ b2: true");
   }
}
```

Результат выполнения данной программы выглядит так.

```
i < j
i <= j
i != j
!(b1 & b2): true
b1 | b2: true
b1 ^ b2: true
```

Укороченные логические операторы

В Java также предусмотрены специальные *укороченные* варианты логических операторов И и ИЛИ, позволяющие выполнять так называемую быструю оценку значений логических выражений, обеспечивающую получение более эффективного кода. Поясним это на следующих примерах. Если первый операнд логического оператора И имеет ложное значение (`false`), то результат будет иметь ложное значение независимо от значения второго операнда. Если же первый операнд логического оператора ИЛИ имеет истинное значение (`true`), то результат будет иметь истинное значение независимо от значения второго операнда. Благодаря тому что значение второго операнда в этих операциях вычислять не нужно, экономится время и повышается эффективность кода.

Укороченному логическому оператору И соответствует обозначение `&&`, а укороченному логическому оператору ИЛИ — обозначение `||`. Аналогичные им обычные логические операции обозначаются знаками `&` и `|`. Единственное отличие укороченного логического оператора от обычного заключается в том, что второй операнд вычисляется только тогда, когда это нужно.

В приведенном ниже примере программы демонстрируется применение укороченного логического оператора И. В этой программе с помощью деления по модулю определяется, делится ли значение переменной `n` на значение переменной `d` нацело. Если остаток от деления `n / d` равен нулю, то `n` делится на `d` нацело. Но поскольку данная операция подразумевает деление, то для предотвращения деления на нуль используется укороченный логический оператор И.

```
// Демонстрация использования укороченных логических операторов
class SCops {
   public static void main(String args[]) {
      int n, d, q;

      n = 10;
      d = 2;
      if(d != 0 && (n % d) == 0)
         System.out.println(d + " является делителем " + n);

      d = 0; // установить для d нулевое значение

      // Второй операнд не вычисляется, поскольку значение
      // переменной d равно нулю
      if(d != 0 && (n % d) == 0)  ◄── Укороченная операция предотвращает деление на нуль
         System.out.println(d + " является делителем " + n);

      // А теперь те же самые действия выполняются без
      // использования укороченного логического оператора.
      // В результате возникнет ошибка деления на нуль.
      if(d != 0 & (n % d) == 0)  ◄── Теперь вычисляются оба выражения, в результате чего будет выполняться деление на нуль
         System.out.println(d + " является делителем " + n);
   }
}
```

С целью предотвращения возможности деления на нуль в условной инструкции `if` сначала проверяется, равно ли нулю значение переменной `d`. Если эта проверка дает истинный результат, вычисление второго операнда укороченного логического оператора И не выполняется. Например, если значение переменной `d` равно 2, то вычисляется остаток от деления по модулю. Если же значение переменной `d` равно нулю, то операция деления по модулю пропускается, чем предотвращается деление на нуль. В конце применяется обычный логический

Спросим у эксперта

ВОПРОС. Если укороченные операторы более эффективны, то почему наряду с ними в Java используются также обычные логические операторы?

ОТВЕТ. В некоторых случаях требуется вычислять оба операнда логического оператора из-за наличия побочных эффектов. Рассмотрим следующий пример.

```
// Демонстрация роли побочных эффектов
class SideEffects {
   public static void main(String args[]) {
      int i;

      i = 0;

      // Значение переменной i инкрементируется несмотря на то,
      // что проверяемое условие в инструкции if ложно
      if(false & (++i < 100))
         System.out.println("Эта строка не будет отображаться");
      System.out.println("Инструкция if выполняется: " + i);
         // отображается 1

      // В данном случае значение переменной i не инкрементируется,
      // поскольку второй операнд укороченного логического
оператора
      // не вычисляется, а следовательно, инкремент пропускается
      if(false && (++i < 100))
         System.out.println("Эта строка не будет отображаться");
      System.out.println("Инструкция if выполняется: " + i);
         // по-прежнему отображается 1!!
   }
}
```

Как следует из приведенного выше фрагмента кода и комментариев к нему, в первой условной инструкции `if` значение переменной `i` должно увеличиваться на единицу, независимо от того, выполняется ли условие этой инструкции. Но когда во второй условной инструкции `if` применяется укороченный логический оператор, значение переменной `i` не инкрементируется, поскольку первый операнд в проверяемом условии имеет логическое значение `false`. Следовательно, если логика программы требует, чтобы второй операнд логического оператора непременно вычислялся, следует применять обычные, а не укороченные формы логических операторов.

оператор И, в котором вычисляются оба операнда, что может привести к делению на нуль при выполнении данной программы.

И последнее замечание: в формальной спецификации Java укороченный оператор И называется *условным логическим оператором И*, а укороченный оператор ИЛИ — *условным логическим оператором ИЛИ*, но чаще всего подобные операторы называют *укороченными*.

Оператор присваивания

Оператор присваивания уже не раз применялся в примерах программ, начиная с главы 1. Теперь настало время дать ему формальное определение. *Оператор присваивания* обозначается одиночным знаком равенства (=). В Java он играет ту же роль, что и в других языках программирования. Общая форма записи этого оператора выглядит так:

```
переменная = выражение
```

где *переменная* и *выражение* должны иметь совместимые типы.

У оператора присваивания имеется одна интересная особенность, о которой вам будет полезно знать: возможность создания цепочки операций присваивания. Рассмотрим, например, следующий фрагмент кода.

```
int x, y, z;
x = y = z = 100; // присвоить значение 100 переменным x, y и z
```

В приведенном выше фрагменте кода одно и то же значение 100 задается для переменных x, y и z с помощью единственного оператора присваивания, в котором значение левого операнда каждой из операций присваивания всякий раз устанавливается равным значению правого операнда. Таким образом, значение 100 присваивается сначала переменной z, затем переменной y и, наконец, переменной x. Такой способ присваивания по цепочке удобен для задания общего значения целой группе переменных.

Составные операторы присваивания

В Java для ряда операций предусмотрены так называемые *составные операторы присваивания*, которые позволяют записывать действие с последующим присвоением в виде одной операции, что делает текст программ более компактным. Обратимся к простому примеру. Приведенный ниже оператор присваивания

```
x = x + 10;
```

можно переписать в более компактной форме:

```
x += 10;
```

Знак операции += указывает компилятору на то, что переменной x должно быть присвоено ее первоначальное значение, увеличенное на 10.

Рассмотрим еще один пример. Операция

```
x = x - 100;
```

и операция

```
x -= 100;
```

выполняют одни и те же действия. И в том и в другом случае переменной x присваивается ее первоначальное значение, уменьшенное на 100.

Для всех бинарных операторов, т.е. таких, которые требуют наличия двух операндов, в Java предусмотрены соответствующие составные операторы присваивания. Общая форма всех этих операторов такова:

переменная операция= выражение

где *операция* — арифметическая или логическая операция, выполняемая совместно с операцией присваивания.

Ниже перечислены составные операторы присваивания для арифметических и логических операций.

+=	-=	*=	/=
%=	&=	|=	^=

Каждый из перечисленных выше операторов представляется знаком соответствующей операции и следующим за ним знаком равенства, что и дало им название "составные".

Составные операторы присваивания обладают двумя главными преимуществами. Во-первых, они записываются в более компактной форме, чем их обычные эквиваленты. Во-вторых, они приводят к более эффективному исполняемому коду, поскольку левый операнд в них вычисляется только один раз. Именно по этим причинам они часто используются в профессионально написанных программах на Java.

Преобразование типов при присваивании

При написании программ очень часто возникает потребность в присваивании значения, хранящегося в переменной одного типа, переменной другого типа. Например, значение int, возможно, потребуется присвоить переменной float.

```
int i;
float f;

i = 10;
f = i; // присвоить значение переменной типа int
       // переменной типа float
```

Если типы данных являются совместимыми, то значение из правой части оператора присваивания автоматически преобразуется в тип данных в его левой части. Так, в приведенном выше фрагменте кода значение переменной i преобразуется в тип float, а затем присваивается переменной f. Но ведь Java — язык со строгим контролем типов, и далеко не все типы данных в нем совместимы,

поэтому неявное преобразование типов выполняется не всегда. В частности, типы `boolean` и `int` несовместимы.

Автоматическое преобразование типов в операции присваивания выполняется при соблюдении следующих условий:

- оба типа являются совместимыми;
- целевой тип обладает более широким диапазоном допустимых значений, чем исходный.

Если соблюдаются оба вышеперечисленных условия, происходит так называемое ***расширение типа***. Например, диапазона значений, допустимых для типа `int`, совершенно достаточно, чтобы представить любое значение типа `byte`, а кроме того, оба этих типа данных являются целочисленными. Поэтому и происходит автоматическое преобразование типа `byte` в тип `int`.

С точки зрения расширения типов целочисленные типы и типы с плавающей точкой совместимы. Например, приведенная ниже программа написана корректно, поскольку преобразование типа `long` в тип `double` является расширяющим и выполняется автоматически.

```
// Демонстрация автоматического преобразования типа long
// в тип double
class LtoD {
   public static void main(String args[]) {
      long L;
      double D;

      L = 100123285L;
      D = L;  ◄——— Автоматическое преобразование типа long в тип double

      System.out.println("L и D: " + L + " " + D);
   }
}
```

В то же время тип `double` не может быть автоматически преобразован в тип `long`, поскольку такое преобразование уже не является расширяющим. Следовательно, приведенный ниже вариант той же самой программы оказывается некорректным.

```
// *** Эта программа не пройдет компиляцию ***
class LtoD {
   public static void main(String args[]) {
      long L;
      double D;

      D = 100123285.0;
      L = D; // Ошибка!!!  ◄——— Тип double не преобразуется автоматически в тип long

      System.out.println("L и D: " + L + " " + D);
   }
}
```

Автоматическое преобразование числовых типов в тип `char` или `boolean` не выполняется. Кроме того, типы `char` и `boolean` несовместимы друг с другом. Тем не менее переменной `char` может быть присвоено значение, представленное целочисленным литералом.

Приведение несовместимых типов

Несмотря на всю полезность неявных автоматических преобразований типов, они не способны удовлетворить все потребности программиста, поскольку допускают лишь расширяющие преобразования совместимых типов. Во всех остальных случаях приходится обращаться к приведению типов. ***Приведение*** — это команда компилятору преобразовать результат вычисления выражения в указанный тип. А для этого требуется явное преобразование типов. Так выглядит общий синтаксис приведения типов:

```
(целевой_тип) выражение
```

где *целевой_тип* обозначает тот тип, в который желательно преобразовать указанное выражение. Так, если требуется привести значение, возвращаемое выражением `x / y`, к типу `int`, это можно сделать следующим образом:

```
double x, y;
// ...
(int) (x / y)
```

В данном случае приведение типов обеспечит преобразование результатов выполнения выражения в тип `int`, несмотря на то что переменные `x` и `y` принадлежат к типу `double`. Выражение `x / y` следует заключить в круглые скобки, иначе будет преобразован не результат деления, а только значение переменной `x`. Приведение типов в данном случае требуется потому, что автоматическое преобразование типа `double` в тип `int` не выполняется.

Если приведение типа означает его ***сужение***, то часть информации может быть утеряна. Например, в результате приведения типа `long` к типу `int` часть информации будет утеряна, если значение типа `long` окажется больше диапазона представления чисел для типа `int`, поскольку старшие разряды этого числового значения отбрасываются. Когда же значение с плавающей точкой приводится к целочисленному значению, в результате усечения теряется дробная часть этого числового значения. Так, если присвоить значение `1.23` целочисленной переменной, то в результате в ней останется лишь целая часть исходного числа (`1`), а дробная часть (`0.23`) будет утеряна.

Ниже приведен пример программы, демонстрирующий некоторые виды преобразований, требующие явного приведения типов.

```
// Демонстрация приведения типов
class CastDemo {
   public static void main(String args[]) {
      double x, y;
      byte b;
```

```
        int i;
        char ch;

        x = 10.0;
        y = 3.0;
                                       В данном случае теряется дробная часть числа
        i = (int) (x / y); // привести тип double к типу int
        System.out.println("Целочисленный результат деления x / y: " + i);

        i = 100;
        b = (byte) i; <---------------------- А в этом случае информация не теряется.
        System.out.println("Значение b: " + b);  Тип byte может содержать значение 100.

        i = 257;
        b = (byte) i; <---------------------- На этот раз информация теряется. Тип
        System.out.println("Значение b: " + b);  byte не может содержать значение 257.

        b = 88; // Представление символа X в коде ASCII
        ch = (char) b;<---------------------- Явное приведение несовместимых типов
        System.out.println("ch: " + ch);
    }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Целочисленный результат деления x / y: 3
Значение b: 100
Значение b: 1
ch: X
```

В данной программе приведение выражения (x / y) к типу int означает потерю дробной части числового значения результата деления. Когда переменной b присваивается значение 100 из переменной i, данные не теряются, поскольку диапазон допустимых значений у типа byte достаточен для представления этого значения. Далее при попытке присвоить переменной b значение 257 снова происходит потеря данных, поскольку значение 257 оказывается за пределами диапазона допустимых значений для типа byte. И наконец, когда переменной char присваивается содержимое переменной типа byte, данные не теряются, но явное приведение типов все же требуется.

Приоритеты операций

В табл. 2.3 приведены приоритеты используемых в Java операторов в порядке следования от самого высокого до самого низкого приоритета. В эту таблицу включен ряд операторов, которые будут рассмотрены в последующих главах. Формально разделители [], () и . могут интерпретироваться как операторы, и в этом случае они будут иметь наивысший приоритет.

Таблица 2.3. Приоритет операций в Java

НАИВЫСШИЙ						
++ (постфиксная)	-- (постфиксная)					
++ (префиксная)	-- (префиксная)	~	!	+ (унарный плюс)	- (унарный минус)	(приведение типов)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
->						
=	*операция*=					
НАИМЕНЬШИЙ						

Упражнение 2.2

Отображение таблицы истинности для логических операций

`LogicalOpTable.java`

В этом упражнении нам предстоит создать программу, которая отображает таблицу истинности для логических операций Java. Для удобства восприятия отображаемой информации следует выровнять столбцы таблицы. В данном примере используется ряд рассмотренных ранее средств языка, включая управляющие последовательности и логические операторы, а также демонстрируются отличия в использовании приоритетов арифметических и логических операторов. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл `LogicalOpTable.java`.
2. Для того чтобы обеспечить выравнивание столбцов таблицы, в каждую выводимую строку следует ввести символы `\t`. В качестве примера ниже приведен вызов метода `println()` для отображения заголовков таблицы.

```
System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");
```

3. Для того чтобы сведения об операторах располагались под соответствующими заголовками, в каждую последующую строку таблицы должны быть введены символы табуляции.

4. **Введите в файл `LogicalOpTable.java` исходный код программы, как показано ниже.**

```
// Упражнение 2.2
// Отображение таблицы истинности для логических операций
class LogicalOpTable {
   public static void main(String args[]) {
      boolean p, q;

      System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");

      p = true; q = true;
      System.out.print(p + "\t" + q +"\t");
      System.out.print((p&q) + "\t" + (p|q) + "\t");
      System.out.println((p^q) + "\t" + (!p));

      p = true; q = false;
      System.out.print(p + "\t" + q +"\t");
      System.out.print((p&q) + "\t" + (p|q) + "\t");
      System.out.println((p^q) + "\t" + (!p));

      p = false; q = true;
      System.out.print(p + "\t" + q +"\t");
      System.out.print((p&q) + "\t" + (p|q) + "\t");
      System.out.println((p^q) + "\t" + (!p));

      p = false; q = false;
      System.out.print(p + "\t" + q +"\t");
      System.out.print((p&q) + "\t" + (p|q) + "\t");
      System.out.println((p^q) + "\t" + (!p));
   }
}
```

Обратите внимание на то, что в инструкциях с вызовами метода `println()` логические операции заключены в круглые скобки. Эти скобки необходимы для соблюдения приоритета операторов. В частности, арифметический оператор + имеет более высокий приоритет, чем логические операторы.

5. Скомпилируйте программу и запустите ее на выполнение. Результат должен выглядеть следующим образом.

```
P       Q       AND     OR      XOR     NOT
true    true    true    true    false   false
true    false   false   true    true    false
false   true    false   true    true    true
false   false   false   false   false   true
```

6. Попытайтесь видоизменить программу так, чтобы вместо логических значений `true` и `false` отображались значения 1 и 0. Это потребует больших усилий, чем кажется на первый взгляд!

Выражения

Операторы, переменные и литералы являются составными частями *выражений*. Выражением в Java может стать любое допустимое сочетание этих элементов. Выражения должны быть уже знакомы вам по предыдущим примерам программ. Более того, вы изучали их в школьном курсе алгебры. Но некоторые их особенности все же нуждаются в обсуждении.

Преобразование типов в выражениях

В выражении можно свободно использовать два или несколько типов данных, при условии их совместимости друг с другом. Например, в одном выражении допускается применение типов short и long, поскольку оба типа являются числовыми. Когда в выражении встречаются разные типы данных, они преобразуются в один общий тип в соответствии с принятыми в Java *правилами повышения типов* (promotion rules).

Сначала все значения типа char, byte и short повышаются до типа int. Затем все выражение повышается до типа long, если хотя бы один из его операндов имеет тип long. Далее все выражение повышается до типа float, если хотя бы один из операндов относится к типу float. А если какой-нибудь из операндов относится к типу double, то результат также относится к типу double.

Очень важно иметь в виду, что правила повышения типов применяются только к значениям, участвующим в вычислении выражений. Например, в то время как значение переменной типа byte при вычислении выражения может повышаться до типа int, за пределами выражения эта переменная будет по-прежнему иметь тип byte. Следовательно, повышение типов применяется только при вычислении выражений.

Но иногда повышение типов может приводить к неожиданным результатам. Если, например, в арифметической операции используются два значения типа byte, то происходит следующее. Сначала операнды типа byte повышаются до типа int, а затем выполняется операция, дающая результат типа int. Следовательно, результат выполнения операции, в которой участвуют два значения типа byte, будет иметь тип int. Но ведь это не тот результат, который можно было бы с очевидностью предположить. Рассмотрим следующий пример программы.

```
// Неожиданный результат повышения типов!
class PromDemo {
   public static void main(String args[]) {
      byte b;
      int i;

      b = 10;
      i = b * b;
```

Приведение типов не требуется, так как тип уже повышен до int

Здесь для присваивания значения int переменной типа byte требуется приведение типов!

```
      b = 10;
      b = (byte) (b * b); // нужно приведение типов!

      System.out.println("i и b: " + i + " " + b);
   }
}
```

Любопытно отметить, что при присваивании выражения b*b переменной i приведение типов не требуется, поскольку тип b автоматически повышается до int при вычислении выражения. В то же время, когда вы пытаетесь присвоить результат вычисления выражения b*b переменной b, требуется выполнить обратное приведение к типу byte! Объясняется это тем, что в выражении b*b значение переменной b повышается до типа int и поэтому не может быть присвоено переменной типа byte без приведения типов. Имейте это обстоятельство в виду, если получите неожиданное сообщение об ошибке несовместимости типов в выражениях, которые на первый взгляд кажутся совершенно правильными.

Аналогичная ситуация возникает при выполнении операций с символьными операндами. Например, в следующем фрагменте кода требуется обратное приведение к типу char, поскольку операнды ch1 и ch2 в выражении повышаются до типа int.

```
char ch1 = 'a', ch2 = 'b';
ch1 = (char) (ch1 + ch2);
```

Без приведения типов результат сложения операндов ch1 и ch2 будет иметь тип int, поэтому его нельзя присвоить переменной типа char.

Приведение типов требуется не только при присваивании значения переменной. Рассмотрим в качестве примера следующую программу. В ней приведение типов выполняется для того, чтобы дробная часть числового значения типа double не была утеряна. В противном случае операция деления будет выполняться над целыми числами.

```
// Приведение типов для правильного вычисления выражения
class UseCast {
   public static void main(String args[]) {
      int i;

      for(i = 0; i < 5; i++) {
         System.out.println(i + " / 3: " + i / 3);
         System.out.println(i + " / 3 с дробной частью: " +
                           (double) i / 3);
         System.out.println();
      }
   }
}
```

Ниже приведен результат выполнения данной программы.

```
0 / 3: 0
0 / 3 с дробной частью: 0.0

1 / 3: 0
1 / 3 с дробной частью: 0.3333333333333333

2 / 3: 0
2 / 3 с дробной частью: 0.6666666666666666

3 / 3: 1
3 / 3 с дробной частью: 1.0

4 / 3: 1
4 / 3 с дробной частью: 1.3333333333333333
```

Пробелы и круглые скобки

Для повышения удобочитаемости выражений в коде Java в них можно использовать символы табуляции и пробелы. Например, ниже приведены два варианта одного и того же выражения, но второй вариант читается гораздо легче.

```
x=10/y*(127/x);

x = 10 / y * (127/x);
```

Круглые скобки повышают приоритет содержащихся в них операторов (аналогичное правило применяется и в алгебре). Избыточные скобки допустимы. Они не приводят к ошибке и не замедляют выполнение программы. В некоторых случаях лишние скобки даже желательны. Они проясняют порядок вычисления выражения как для вас, так и для тех, кто будет разбирать исходный код вашей программы. Какое из приведенных ниже двух выражений воспринимается легче?

```
x = y/3-34*temp+127;

x = (y/3) - (34*temp) + 127;
```

Вопросы и упражнения для самопроверки

1. Почему в Java строго определены диапазоны допустимых значений и области действия простых типов?
2. Что собой представляет символьный тип в Java и чем он отличается от символьного типа в ряде других языков программирования?
3. Переменная типа `boolean` может иметь любое значение, поскольку любое ненулевое значение интерпретируется как истинное. Верно или неверно?

4. Допустим, результат выполнения программы выглядит следующим образом.

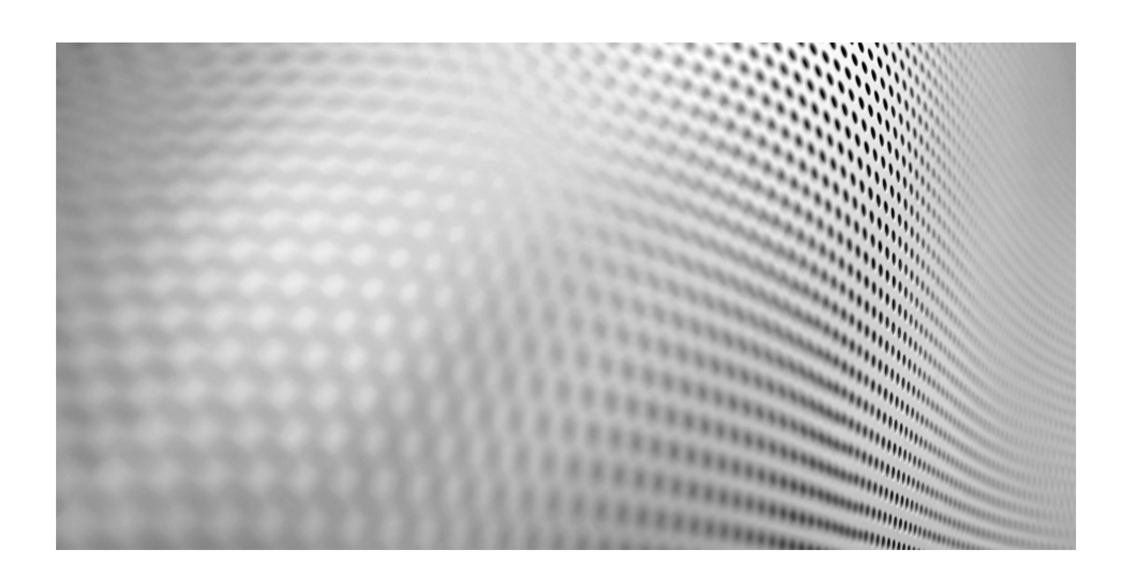
```
Один
Два
Три
```

Напишите строку кода с вызовом метода `println()`, где этот результат выводится в виде одной строки.

5. Какая ошибка допущена в следующем фрагменте кода?

```
for(i = 0; i < 10; i++) {
   int sum;

   sum = sum + i;
}
System.out.println("Сумма: " + sum);
```

6. Поясните различие между префиксной и постфиксной формами записи оператора инкремента.
7. Покажите, каким образом укороченный логический оператор И может предотвратить деление на нуль.
8. До какого типа повышаются типы `byte` и `short` при вычислении выражения?
9. Когда возникает потребность в явном приведении типов?
10. Напишите программу, которая находила бы все простые числа в диапазоне от 2 до 100.
11. Влияют ли лишние скобки на эффективность выполнения программ?
12. Определяет ли блок кода область действия переменных?



Глава 3

Управляющие инструкции

В этой главе...

- Ввод символов с клавиатуры
- Полная форма условной инструкции `if`
- Инструкция `switch`
- Полная форма цикла `for`
- Цикл `while`
- Цикл `do-while`
- Использование инструкции `break` для выхода из цикла
- Использование инструкции `break` в качестве оператора `goto`
- Инструкция `continue`
- Вложенные циклы

В этой главе вы познакомитесь с инструкциями, которые управляют выполнением программы. Существуют три категории управляющих инструкций: инструкции *выбора*, к числу которых относятся `if` и `switch`, *итерационные* инструкции, в том числе циклы `for`, `while`, `do-while`, а также инструкции *перехода*, включая `break`, `continue` и `return`. Все эти управляющие инструкции, кроме `return`, рассматриваемой позже, подробно описаны в данной главе, в начале которой будет показано, каким образом выполняется простой ввод данных с клавиатуры.

Ввод символов с клавиатуры

Прежде чем приступать к рассмотрению управляющих инструкций в Java, уделим немного внимания средствам, которые позволяют создавать интерактивные программы. В рассмотренных до сих пор примерах программ данные выводились на экран, но у пользователя не было возможности вводить данные. В этих программах, в частности, применялся консольный вывод, но не консольный ввод (с клавиатуры). И объясняется это тем, что возможности ввода данных с клавиатуры в Java опираются на языковые средства, рассматриваемые в последующих главах. Кроме того, большинство реальных приложений Java имеют графический и оконный, а не консольный интерфейс. Именно по этим причинам консольный ввод нечасто применяется в примерах программ, представленных в данной книге. Но имеется один вид консольного ввода, который реализуется очень просто: это чтение символов с клавиатуры. А поскольку ввод символов применяется в ряде примеров, представленных в данной главе, мы и начнем ее с обсуждения данного вопроса.

Для чтения символа с клавиатуры достаточно вызвать метод `System.in.read()`, где `System.in` — объект ввода (с клавиатуры), дополняющий объект вывода `System.out`. Метод `read()` ожидает нажатия пользователем какой-либо клавиши, после чего возвращает результат. Возвращаемый им символ представлен целочисленным значением, и поэтому, прежде чем присвоить его символьной переменной, следует явно привести его к типу `char`. По умолчанию данные, вводимые с консоли, *буферизуются построчно*. Под термином *буфер* здесь подразумевается небольшая область памяти, выделяемая для хранения символов перед тем, как они будут прочитаны программой. В данном случае в буфере хранится целая текстовая строка, и поэтому для передачи программе любого введенного с клавиатуры символа следует нажать клавишу <Enter>. Ниже приведен пример программы, которая читает символы, вводимые с клавиатуры.

```
// Чтение символа с клавиатуры
class KbIn {
   public static void main(String args[])
      throws java.io.IOException {

      char ch;

      System.out.print("Нажмите нужную клавишу, а затем
                        клавишу ENTER: ");
      ch = (char) System.in.read(); // получить символ  ◄—— Ввод символа с клавиатуры

      System.out.println("Вы нажали клавишу " + ch);
   }
}
```

Выполнение этой программы может дать, например, следующий результат.

```
Нажмите нужную клавишу, а затем клавишу ENTER: t
Вы нажали клавишу t
```

Обратите внимание на то, что метод `main()` начинается со следующих строк кода.

```
public static void main(String args[])
   throws java.io.IOException {
```

В рассматриваемой здесь программе применяется метод `System.in.read()`, и поэтому в ее код следует добавить выражение `throws java.io.IOException`. Эта инструкция требуется для обработки ошибок, которые могут возникнуть в процессе ввода данных. Она является частью механизма обработки исключений в Java, более подробно рассматриваемого в главе 9. А до тех пор можете не обращать на нее особого внимания, просто помните о ее назначении.

Построчная буферизация вводимых данных средствами `System.in` часто приводит к недоразумениям. При нажатии клавиши <Enter> в поток ввода записывается последовательность, состоящая из символов возврата каретки и перевода строки. Эти символы ожидают чтения из буфера ввода. Поэтому в некоторых приложениях, возможно, потребуется удалить символы возврата каретки

и перевода строки, прежде чем переходить к следующей операции ввода. Для этого достаточно прочитать их из буфера ввода. Соответствующий пример реализации подобного решения на практике будет представлен далее.

Условная инструкция if

Эта условная инструкция уже была представлена в главе 1, а сейчас она будет рассмотрена более подробно. Условные инструкции называют также командами ветвления, поскольку с их помощью выбирается ветвь кода, подлежащая выполнению. Ниже приведена полная форма условной инструкции `if`.

```
if(условие) инструкция;
else инструкция;
```

Здесь *условие* — это некоторое условное выражение, а после ключевого слова `if` или `else` стоит одиночная инструкция. Предложение `else` не является обязательным. После ключевых слов `if` и `else` могут также стоять блоки инструкций. Ниже приведена общая форма условной инструкции `if`, в которой используются блоки кода.

```
if(условие)
{
   последовательность инструкций
}
else
{
   последовательность инструкций
}
```

Если условное выражение оказывается истинным, то выполняется ветвь `if`. В противном случае выполняется ветвь `else`, если таковая существует. Выполнение сразу двух ветвей невозможно. Условное выражение, управляющее инструкцией `if`, должно давать результат типа `boolean`.

Для того чтобы продемонстрировать применение инструкции `if` (и ряда других управляющих инструкций) на практике, разработаем простую игру, основанную на угадывании. Возможно, она понравится вашим детям. В первой версии этой игры программа предложит пользователю угадать задуманную букву от 'A' до 'Z'. Если пользователь правильно угадает букву и нажмет на клавиатуре соответствующую клавишу, то программа выведет сообщение `**Правильно!**`. Ниже приведен исходный код программы, реализующей эту игру.

```
// Игра в угадывание букв
class Guess {
   public static void main(String args[])
      throws java.io.IOException {

      char ch, answer = 'K';

      System.out.println("Задумана буква из диапазона A-Z.");
      System.out.print("Попытайтесь ее угадать: ");
```

```
      ch = (char) System.in.read(); // чтение символа с клавиатуры

      if(ch == answer) System.out.println("** Правильно! **");
   }
}
```

Эта программа выводит на экран сообщение с предложением угадать букву, а затем читает символ с клавиатуры. Используя условную инструкцию `if`, она сравнивает введенный символ с правильным вариантом (в данном случае это буква 'K'). Если введена буква 'K', то отображается сообщение о том, что ответ правильный. Работая с этой программой, не забывайте, что угадываемую букву следует вводить в верхнем регистре.

В следующей версии программы ветвь `else` используется для вывода сообщения о том, что буква не угадана.

```
// Игра в угадывание букв, вторая версия
// class Guess2 {
   public static void main(String args[])
      throws java.io.IOException {

      char ch, answer = 'K';

      System.out.println("Задумана буква из диапазона A-Z.");
      System.out.print("Попытайтесь ее угадать: ");

      ch = (char) System.in.read(); // чтение символа с клавиатуры

      if(ch == answer) System.out.println("** Правильно! **");
      else System.out.println("...Извините, вы не угадали.");
   }
}
```

Вложенные условные инструкции if

Вложенные инструкции `if` представляют собой условные инструкции, являющиеся телом ветви `if` или `else`. Подобные условные инструкции очень часто встречаются в программах. Но, пользуясь ими, следует помнить, что в Java ветвь `else` всегда связана с ближайшей к ней ветви `if`, находящейся в том же блоке кода и не связанной с другим предложением `else`. Рассмотрим пример.

```
if(i == 10) {
   if(j < 20) a = b;
   if(k > 100) c = d;
   else a = c; // это предложение else относится
               // к предложению if(k > 100)
}
else a = d;    // а это предложение else относится
               // к предложению if(i == 10)
```

Как следует из комментариев к приведенному выше фрагменту кода, последнее предложение `else` не имеет отношения к предложению `if(j < 20)`,

поскольку оно не находится с ним в одном блоке, несмотря на то что это ближайшая ветвь `if`, не имеющая парной ветви `else`. Следовательно, последнее предложение `else` относится к предложению `if(i == 10)`. А предложение `else`, находящееся в блоке, связано с предложением `if(k > 100)`, поскольку это самая близкая из всех находящихся к нему ветвей `if` в том же самом блоке.

Используя вложенные условные инструкции `if`, можно усовершенствовать игру, рассматриваемую здесь в качестве примера. Теперь при неудачной попытке угадать букву пользователю предоставляется дополнительная информация, подсказывающая, насколько сильно предлагаемый вариант отличается от правильного ответа.

```
// Игра в угадывание букв, третья версия
class Guess3 {
   public static void main(String args[])
      throws java.io.IOException {

      char ch, answer = 'K';

      System.out.println("Задумана буква из диапазона A-Z.");
      System.out.print("Попытайтесь ее угадать: ");

      ch = (char) System.in.read(); // чтение символа с клавиатуры

      if(ch == answer) System.out.println("** Правильно! **");
      else {
         System.out.print("...Извините, нужная буква находится ");

         // вложенная инструкция if
         if(ch < answer) System.out.println("ближе к концу алфавита");
         else System.out.println("ближе к началу алфавита");
      }
   }
}
```

Вложенная инструкция if

В результате выполнения этой программы может быть получен следующий результат.

```
Задумана буква из диапазона A-Z.
Попытайтесь ее угадать: Z
...Извините, нужная буква находится ближе к началу алфавита.
```

Многоступенчатая конструкция if-else-if

В программировании часто применяется *многоступенчатая* конструкция `if-else-if`, состоящая из вложенных условных инструкций `if`. Ниже приведен ее общий синтаксис.

```
if(условие)
   инструкция;
else if (условие)
   инструкция;
```

```
else if (условие)
   инструкция;
.
.
.
else
   инструкция;
```

Условные выражения в такой конструкции вычисляются в направлении сверху вниз. Как только обнаружится истинное условие, выполняется связанная с ним инструкция, а все остальные блоки многоступенчатой конструкции опускаются. Если ни одно из условий не является истинным, то выполняется последняя ветвь `else`, которая зачастую играет роль условия, выбираемого по умолчанию. Когда же последняя ветвь `else` отсутствует, а все остальные проверки по условию дают ложный результат, никаких действий вообще не выполняется.

Ниже приведен пример программы, демонстрирующий применение многоступенчатой конструкции `if-else-if`.

```
// Демонстрация использования многоступенчатой
// конструкции if-else-if
class Ladder {
   public static void main(String args[]) {
      int x;

      for(x=0; x<6; x++) {
         if(x==1)
            System.out.println("x равно 1");
         else if(x==2)
            System.out.println("x равно 2");
         else if(x==3)
            System.out.println("x равно 3");
         else if(x==4)
            System.out.println("x равно 4");
         else
            // Условие, выполняемое по умолчанию
            System.out.println("Значение x находится вне
                                диапазона 1-4"); ◄—— Оператор, заданный по умолчанию
      }
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Значение x находится вне диапазона 1-4
x равно 1
x равно 2
x равно 3
x равно 4
Значение x находится вне диапазона 1-4
```

Как видите, устанавливаемая по умолчанию ветвь `else` выполняется лишь в том случае, если проверка условий каждого из предыдущих предложений `if` дает ложный результат.

Инструкция switch

Второй инструкцией выбора в Java является `switch`, которая обеспечивает многовариантное ветвление программы. Эта инструкция позволяет сделать выбор среди нескольких альтернативных вариантов (ветвей) дальнейшего выполнения программы. Несмотря на то что многовариантная проверка может быть организована с помощью последовательного ряда вложенных условных инструкций `if`, во многих случаях более эффективным оказывается применение инструкции `switch`. Она действует следующим образом. Значение выражения последовательно сравнивается с константами из заданного списка. Как только будет обнаружено совпадение, выполняется соответствующая последовательность инструкций. Ниже приведен общий синтаксис инструкции `switch`.

```
switch(выражение) {
   case константа1:
      последовательность инструкций
      break;
   case константа2:
      последовательность инструкций
      break;
   case константа3:
      последовательность инструкций
      break;
.
.
.
   default:
      последовательность инструкций
}
```

В версиях Java, предшествующих JDK 7, *выражение*, управляющее инструкцией `switch`, должно быть типа `byte`, `short`, `int`, `char` или перечислением. (Подробнее перечисления будут рассмотрены в главе 12.)

Начиная с версии JDK 7 *выражение* может относиться к типу `String`. Это означает, что в современных версиях Java для управления инструкцией `switch` можно пользоваться символьной строкой. (Этот прием программирования будет продемонстрирован в главе 5 при рассмотрении класса `String`.) Но зачастую в качестве выражения, управляющего инструкцией `switch`, вместо сложного выражения используется простая переменная.

Каждое значение, указанное в предложениях `case`, должно быть уникальным выражением, включающим константы (например, значением литерала). Не допускаются дубликаты значений `case`. Тип каждого значения должен быть совместимым с типом выражения.

Последовательность инструкций из ветви `default` выполняется в том случае, если ни одна из констант выбора, находящихся в ветви `case`, не совпадает с заданным выражением. Ветвь `default` не является обязательной. Если она отсутствует и выражение не совпадает ни с одним из условий выбора, то никаких действий вообще не выполняется. Если же имеет место совпадение с одним из условий выбора, то выполняются инструкции, связанные с этим условием, вплоть до инструкции `break`. Либо, если это предложение `default` либо последнее предложение `case`, выполняется все, вплоть до конца инструкции `switch`.

Ниже приведен пример программы, демонстрирующий применение инструкции `switch`.

```
// Демонстрация использования инструкции switch
class SwitchDemo {
   public static void main(String args[]) {
      int i;

      for(i=0; i<10; i++)
         switch(i) {
            case 0:
               System.out.println("i равно 0");
               break;
            case 1:
               System.out.println("i равно 1");
               break;
            case 2:
               System.out.println("i равно 2");
               break;
            case 3:
               System.out.println("i равно 3");
               break;
            case 4:
               System.out.println("i равно 4");
               break;
            default:
               System.out.println("i равно или больше 5");
         }
      }
}
```

Результат выполнения данной программы будет следующим.

```
i равно 0
i равно 1
i равно 2
i равно 3
i равно 4
i равно или больше 5
i равно или больше 5
i равно или больше 5
i равно или больше 5
i равно или больше 5
```

Как видите, на каждой итерации цикла выполняются инструкции, связанные с совпадающей константой выбора, находящейся в одной из ветвей case, в обход всех остальных ветвей. Когда же значение переменной i становится равным пяти или больше, оно не совпадает ни с одной из констант выбора, и поэтому управление получает инструкция, следующая за предложением default.

Формально инструкция break может отсутствовать, но, как правило, в реальных приложениях она все же применяется. При выполнении инструкции break инструкция switch завершает работу, и управление передается следующей за ней инструкции. Если же в последовательности инструкций, связанных с совпадающей константой выбора в одной из ветвей case, не содержится инструкция break, то сначала выполняются все инструкции в этой ветви, а затем инструкции следующей ветви case. Этот процесс продолжается до тех пор, пока не встретится инструкция break или же конец инструкции switch.

В качестве упражнения проанализируйте исходный код приведенной ниже программы. Сможете ли вы предсказать, как будет выглядеть результат ее выполнения?

```
// Демонстрация использования инструкции switch без break
class NoBreak {
   public static void main(String args[]) {
      int i;

      for(i=0; i<=5; i++) {
         switch(i) {
            case 0:
               System.out.println("i меньше 1");
            case 1:
               System.out.println("i меньше 2");
            case 2:
               System.out.println("i меньше 3");
            case 3:
               System.out.println("i меньше 4");
            case 4:
               System.out.println("i меньше 5");
         }
         System.out.println();
      }
   }
}
```

"Проваливание" потока выполнения сквозь ветви case

В результате выполнения этой программы будет получен следующий результат.

```
i меньше 1
i меньше 2
i меньше 3
i меньше 4
i меньше 5

i меньше 2
i меньше 3
```

```
i меньше 4
i меньше 5

i меньше 3
i меньше 4
i меньше 5

i меньше 4
i меньше 5

i меньше 5
```

Как демонстрирует приведенный выше пример, в случае отсутствия инструкции `break` выполнение программы продолжается в следующей ветви `case`. А в следующем примере кода показано, что в инструкции `switch` могут быть пустые ветви `case`.

```
switch(i) {
   case 1:
   case 2:
   case 3: System.out.println("i равно 1, 2 или 3");
      break;
   case 4: System.out.println("i равно 4");
      break;
}
```

Если в приведенном выше фрагменте кода переменная `i` имеет значение 1, 2 или 3, то вызывается первый метод `println()`. А если ее значение равно 4, то вызывается второй метод `println()`. Подобное расположение нескольких пустых ветвей `case` подряд нередко используется тогда, когда нескольким ветвям должен соответствовать один и тот же общий код.

Вложенные инструкции switch

Одна инструкция `switch` может входить в последовательность инструкций другой, внешней инструкции `switch`. Подобная инструкция `switch` называется *вложенной*. Константы выбора внутренней и внешней инструкции `switch` могут содержать общие значения, не вызывая при этом каких-либо конфликтов. Например, следующий фрагмент кода вполне допустим.

```
switch(ch1) {
   case 'A':
      System.out.println("Это ветвь внешней инструкции switch");
      switch(ch2) {
         case 'A':
            System.out.println("Это ветвь внутренней
                               инструкции switch");
            break;
         case 'B': // ...
      } // конец внутренней инструкции switch
      break;
   case 'B': // ...
```

Упражнение 3.1

Начало построения справочной системы Java

Help.java

В этом упражнении будет создана простая справочная система, предоставляющая сведения о синтаксисе управляющих инструкций Java. Программа, реализующая эту справочную систему, отображает меню с названиями инструкций и ожидает выбора одной из них. Как только пользователь выберет один из пунктов меню, на экране отобразятся сведения о синтаксисе соответствующей инструкции. В первой версии данной программы предоставляются сведения только об инструкциях `if` и `switch`, а в последующих упражнениях будут добавлены справочные сведения об остальных управляющих инструкциях. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл `Help.java`.
2. В начале работы программы отображается следующее меню.

```
Справка:
  1. if
  2. switch
Выберите:
```

Это меню реализовано с помощью приведенной ниже последовательности инструкций.

```
System.out.println("Справка:");
System.out.println("  1. if");
System.out.println("  2. switch");
System.out.print("Выберите: ");
```

3. Программа получает данные о варианте, выбранном пользователем. С этой целью вызывается метод `System.in.read()`:

```
choice = (char) System.in.read();
```

4. Для отображения сведений о синтаксисе выбранной инструкции в программе используется инструкция `switch`.

```
switch(choice) {
   case '1':
      System.out.println("Инструкция if:\n");
      System.out.println("if(условие) инструкция;");
      System.out.println("else инструкция;");
      break;
   case '2':
      System.out.println("Инструкция switch:\n");
      System.out.println("switch(выражение) {");
      System.out.println(" case константа:");
      System.out.println(" последовательность инструкций");
      System.out.println(" break;");
      System.out.println(" // ...");
      System.out.println("}");
```

```
      break;
   default:
      System.out.print("Запрос не найден.");
}
```

Обратите внимание на то, как в ветви `default` перехватываются сведения о неправильно сделанном выборе. Так, если пользователь введет значение **3**, оно не совпадет ни с одной из констант в ветвях `case` инструкции `switch`, и управление будет передано коду в ветви `default`.

5. Ниже приведен весь исходный код программы из файла `Help.java`.

```
/*
   Упражнение 3.1

   Простая справочная система.
*/
class Help {
   public static void main(String args[])
      throws java.io.IOException {

      char choice;

      System.out.println("Справка:");
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.print("Выберите: ");
      choice = (char) System.in.read();

      System.out.println("\n");

      switch(choice) {
         case '1':
            System.out.println("Инструкция if:\n");
            System.out.println("if(условие) инструкция;");
            System.out.println("else инструкция;");
            break;
         case '2':
            System.out.println("Инструкция switch:\n");
            System.out.println("switch(выражение) {");
            System.out.println("  case константа:");
            System.out.println("    последовательность инструкций");
            System.out.println("    break;");
            System.out.println("  // ...");
            System.out.println("}");
            break;
         default:
            System.out.print("Запрос не найден.");
      }
   }
}
```

6. В результате выполнения этой программы будет получен следующий результат.

```
Справка:
  1. if
  2. switch
Выберите: 1

Инструкция if:

if(условие) инструкция;
else инструкция;
```

Спросим у эксперта

ВОПРОС. В каких случаях вместо инструкции `switch` следует использовать многоступенчатую конструкцию `if-else-if`?

ОТВЕТ. Вообще говоря, многоступенчатая конструкция `if-else-if` уместна в тех случаях, когда проверка условий не сводится к выяснению совпадения или несовпадения выражения с одиночным значением. Рассмотрим для примера следующую последовательность условных инструкций.

```
if(x < 10) // ...
else if(y != 0) // ...
   else if(!done) // ...
```

Данную последовательность нельзя заменить инструкцией `switch`, поскольку в трех ее условиях используются разные переменные и разные виды сравнения. Многоступенчатую конструкцию `if-else-if` целесообразно также применять для проверки значений с плавающей точкой и других объектов, типы которых отличаются от типов, предусмотренных для ветвей инструкции `switch`.

Цикл for

Цикл `for` уже был представлен в главе 1, здесь же он рассматривается более подробно. Держу пари, что вас приятно удивит эффективность и гибкость этого цикла. Прежде всего обратимся к самым основным и традиционным формам цикла `for`.

Ниже приведен общий синтаксис цикла `for`, применяемого для повторного выполнения единственной инструкции.

```
for(инициализация; условие; итерация) инструкция;
```

А так выглядит его синтаксис для повторного выполнения блока кода.

```
for(инициализация; условие; итерация)
{
   последовательность инструкций;
}
```

Здесь *инициализация*, как правило, представлена оператором присваивания, задающим первоначальное значение управляющей переменной, которая играет роль счетчика цикла; *условие* — это логическое выражение, определяющее необходимость повторения цикла; а *итерация* — это выражение, определяющее величину, на которую должно изменяться значение управляющей переменной (на каждом шаге цикла). Обратите внимание на то, что эти три составные части определения цикла `for` должны быть разделены точкой с запятой. Выполнение цикла `for` будет продолжаться до тех пор, пока результат проверки условия будет истинным. Как только проверка даст ложный результат, цикл завершится, а выполнение программы будет продолжено с инструкции, следующей после цикла `for`.

А теперь рассмотрим пример программы, где цикл `for` служит для вывода на экран значений квадратного корня чисел от 1 до 99. В данной программе отображается также ошибка округления, допущенная при вычислении квадратного корня.

```
// Вывод квадратных корней чисел от 1 до 99
// вместе с ошибкой округления
class SqrRoot {
   public static void main(String args[]) {
      double num, sroot, rerr;

      for(num = 1.0; num < 100.0; num++) {
         sroot = Math.sqrt(num);
         System.out.println("Корень квадратный из " + num +
                            " равен " + sroot);

         // Вычисление ошибки округления
         rerr = num - (sroot * sroot);
         System.out.println("Ошибка округления: " + rerr);
         System.out.println();
      }
   }
}
```

Обратите внимание на то, что ошибка округления вычисляется путем возведения в квадрат квадратного корня числа. Полученное значение вычитается из исходного числа.

Значение переменной цикла может увеличиваться либо уменьшаться, а величина приращения может выбираться произвольно. Например, в приведенном ниже фрагменте кода выводятся числа от 100 до −95, и на каждом шаге значение переменной цикла уменьшается на 5.

```
// Цикл for, выполняющийся с отрицательным приращением переменной
class DecrFor {
   public static void main(String args[]) {
      int x;
```

```
      for(x = 100; x > -100; x -= 5)  ◄—— На каждой итерации переменная
        System.out.println(x);             цикла уменьшается на 5
    }
}
```

В отношении циклов for следует особо подчеркнуть, что условие всегда проверяется в самом начале цикла. Это означает, что код в цикле может вообще не выполняться, если проверяемое условие с самого начала оказывается ложным. Рассмотрим следующий пример.

```
for(count=10; count < 5; count++)
  x += count; // эта инструкция не будет выполнена
```

Этот цикл вообще не будет выполняться, поскольку первоначальное значение переменной count, которая им управляет, сразу же оказывается больше 5. А это означает, что условное выражение count < 5 оказывается ложным с самого начала, т.е. еще до выполнения первого шага цикла.

Некоторые разновидности цикла for

Цикл for относится к наиболее универсальным средствам языка Java, поскольку он допускает самые разные варианты применения. Например, для управления циклом можно использовать несколько переменных. Рассмотрим следующий пример программы.

```
// Применение запятых в определении цикла for
class Comma {
  public static void main(String args[]) {
    int i, j;

    for(i=0, j=10; i < j; i++, j--) ◄—— Для управления этим
      System.out.println("i и j: " + i + " " + j);   циклом используются
  }                                                  две переменные
}
```

В результате выполнения этой программы будет получен следующий результат.

```
i и j: 0 10
i и j: 1 9
i и j: 2 8
i и j: 3 7
i и j: 4 6
```

В данном примере запятыми разделяются два оператора инициализации и еще два итерационных выражения. Когда цикл начинается, инициализируются обе переменные, i и j. Всякий раз, когда цикл повторяется, переменная i инкрементируется, а переменная j декрементируется. Применение нескольких переменных управления циклом нередко оказывается удобным и упрощает некоторые алгоритмы. Теоретически в цикле for может быть указано любое количество выражений инициализации и итерации, но на практике такой цикл получается слишком громоздким, если применяется более двух подобных выражений.

Условием цикла for может быть любое действительное выражение, дающее результат типа boolean. Причем это выражение не обязательно должно включать переменную управления циклом. В следующем примере цикл будет выполняться до тех пор, пока пользователь не введет с клавиатуры букву **S**.

```
// Выполнение цикла до тех пор, пока с клавиатуры
// не будет введена буква S
class ForTest {
   public static void main(String args[])
      throws java.io.IOException {

      int i;

      System.out.println("Для остановки нажмите клавишу S");

      for(i = 0; (char) System.in.read() != 'S'; i++)
         System.out.println("Проход #" + i);
   }
}
```

Пропуск отдельных частей в определении цикла for

Ряд интересных разновидностей цикла for получается в том случае, если оставить пустыми отдельные части в определении цикла. В Java допускается оставлять пустыми любые или же все выражения инициализации, условия и итерации в определении цикла for. В качестве примера рассмотрим следующую программу.

```
// Пропуск отдельных составляющих в определении цикла for
class Empty {
   public static void main(String args[]) {
      int i;

      for(i = 0; i < 10; ) { ◄———— Отсутствует итерационное выражение
         System.out.println("Проход #" + i);
         i++; // инкрементирование переменной цикла
      }
   }
}
```

В данном примере итерационное выражение в определении цикла for оказывается пустым, т.е. вообще отсутствует. Вместо этого переменная i, управляющая циклом, инкрементируется в теле самого цикла. Это означает, что всякий раз, когда цикл повторяется, значение переменной i проверяется на равенство числу 10, но никаких других действий при этом не происходит. А поскольку переменная i инкрементируется в теле цикла, то сам цикл выполняется обычным образом, выводя такие результаты.

```
Проход #0
Проход #1
Проход #2
Проход #3
Проход #4
Проход #5
Проход #6
Проход #7
Проход #8
Проход #9
```

В следующем примере программы из определения цикла `for` исключена инициализирующая часть.

```
// Пропуск отдельных составляющих в определении цикла for
class Empty2 {
   public static void main(String args[]) {
      int i;

      i = 0; // Выносим инициализацию за пределы цикла
      for(; i < 10; ) {
         System.out.println("Проход #" + i);
         i++; // Инкрементирование переменной цикла
      }
   }
}
```

Из определения цикла исключено инициализирующее выражение

В данном примере переменная `i` инициализируется перед началом цикла, а не в самом цикле `for`. Как правило, переменная цикла инициализируется в цикле `for`. Выведение инициализирующей части за пределы цикла обычно делается лишь в том случае, если первоначальное значение управляющей им переменной получается в результате сложного процесса, который нецелесообразно вводить в само определение цикла `for`.

Бесконечный цикл

Если оставить пустым выражение условия в определении цикла `for`, то получится *бесконечный цикл*, т.е. такой, который никогда не завершается. В качестве примера в следующем фрагменте кода показано, каким образом в Java обычно создается бесконечный цикл.

```
for(;;) // этот цикл намеренно сделан бесконечным
{
   //...
}
```

Этот цикл будет выполняться бесконечно. Несмотря на то что бесконечные циклы требуются для решения некоторых задач программирования, например при разработке командных процессоров операционных систем, большинство так называемых "бесконечных" циклов на самом деле представляют собой циклы со специальными требованиями к завершению. (Далее мы поговорим об этом более подробно, однако уже сейчас можно заметить, что в

большинстве случаев выход из бесконечного цикла осуществляется с помощью инструкции `break`.)

Циклы без тела

В Java допускается оставлять пустым тело цикла `for` или любого другого цикла, поскольку *пустая инструкция* с точки зрения синтаксиса этого языка считается действительной. Циклы без тела нередко оказываются полезными. Например, в следующей программе цикл без тела служит для получения суммы чисел от 1 до 5.

```
// Тело цикла for может быть пустым
class Empty3 {
   public static void main(String args[]) {
      int i;
      int sum = 0;

      // суммируются числа от 1 до 5
      for(i = 1; i <= 5; sum += i++);   ◄——— В цикле отсутствует тело!

      System.out.println("Сумма: " + sum);
   }
}
```

В результате выполнения этой программы будет получен следующий результат:

```
Сумма: 15
```

Обратите внимание на то, что суммирование чисел выполняется полностью в определении цикла `for`, и для этого тело цикла не требуется. В данном цикле особое внимание обращает на себя итерационное выражение:

```
sum += i++
```

Подобные операторы не должны вас смущать. Они часто встречаются в программах на Java и становятся вполне понятными, если разобрать их по частям. Приведенный выше оператор означает буквально следующее: сложить со значением переменной `sum` результат суммирования значений переменных `sum` и `i`, после чего инкрементировать значение переменной `i`. Следовательно, данный оператор равнозначен следующей последовательности инструкций.

```
sum = sum + i;
i++;
```

Объявление управляющих переменных в цикле `for`

Нередко переменная цикла `for` требуется только для выполнения самого цикла и нигде больше не используется. В таком случае ее можно объявить в инициализирующей части определения цикла `for`. Например, в приведенной

ниже программе вычисляются сумма и факториал чисел от 1 до 5, а переменная i, управляющая циклом for, объявляется в самом цикле.

```
// Объявление переменной цикла в самом цикле for
class ForVar {
   public static void main(String args[]) {
      int sum = 0;
      int fact = 1;

      // Вычисление факториала чисел от 1 до 5
      for(int i = 1; i <= 5; i++) {  ◄── Переменная i объявляется в самом цикле for
         sum += i; // переменная i доступна во всем цикле
         fact *= i;
      }

      // однако здесь переменная i недоступна

      System.out.println("Сумма: " + sum);
      System.out.println("Факториал: " + fact);
   }
}
```

Объявляя переменную в цикле for, не следует забывать о том, что область действия этой переменной ограничивается определением цикла for. Это означает, что за пределами цикла действие данной переменной прекращается. Так, в приведенном выше примере переменная i оказывается недоступной за пределами цикла for. Для того чтобы использовать переменную цикла в каком-нибудь другом месте программы, ее нельзя объявлять в цикле for.

Прежде чем переходить к последующим разделам, поэкспериментируйте самостоятельно с разновидностями цикла for. В ходе эксперимента вы непременно откроете для себя достоинства этого цикла.

Расширенный цикл for

С недавних пор в распоряжение программистов на Java предоставлен так называемый "расширенный" цикл for, обеспечивающий специальные средства для перебора объектов из коллекции, например из массива. Расширенный цикл for будет рассмотрен в главе 5 применительно к массивам.

Цикл while

Еще одной разновидностью циклов в Java является цикл while. Ниже приведен общий синтаксис этого цикла:

```
while (условие) блок;
```

где *блок* — это одиночная инструкция или блок инструкций, а *условие* означает конкретное условие управления циклом и может быть любым логическим выражением. В этом цикле блок выполняется до тех пор, пока условие истинно.

Как только условие становится ложным, управление программой передается строке кода, следующей непосредственно после цикла.

Ниже приведен простой пример использования цикла while для вывода на экран букв английского алфавита.

```
// Демонстрация использования цикла while
class WhileDemo {
   public static void main(String args[]) {
      char ch;

      // вывод букв английского алфавита, используя цикл while
      ch = 'a';
      while(ch <= 'z') {
         System.out.print(ch);
         ch++;
      }
   }
}
```

В данном примере переменная ch инициализируется кодом буквы 'a'. На каждом шаге цикла на экран сначала выводится значение переменной ch, а затем это значение увеличивается на единицу. Процесс продолжается до тех пор, пока значение переменной ch не станет больше кода буквы 'z'.

Как и в цикле for, в цикле while проверяется условное выражение, указываемое в самом начале цикла. Это означает, что код в теле цикла может вообще не выполняться, а кроме того, избавляет от необходимости выполнять отдельную проверку перед самим циклом. Данное свойство цикла while демонстрируется в следующем примере программы, где вычисляются целые степени числа 2: от 2^0 до 2^9.

```
// Вычисление целых степеней числа 2
class Power {
   public static void main(String args[]) {
      int e;
      int result;

      for(int i=0; i < 10; i++) {
         result = 1;
         e = i;
         while(e > 0) {
            result *= 2;
            e--;
         }

         System.out.println("2 в степени " + i +
                            " равно " + result);
      }
   }
}
```

Ниже приведен результат выполнения данной программы.

```
2 в степени 0 равно 1
2 в степени 1 равно 2
2 в степени 2 равно 4
2 в степени 3 равно 8
2 в степени 4 равно 16
2 в степени 5 равно 32
2 в степени 6 равно 64
2 в степени 7 равно 128
2 в степени 8 равно 256
2 в степени 9 равно 512
```

Обратите внимание на то, что цикл `while` выполняется только в том случае, если значение переменной `e` больше нуля. А когда оно равно нулю, как это имеет место на первом шаге цикла `for`, цикл `while` пропускается.

Спросим у эксперта

ВОПРОС. В языке Java циклы обладают большой гибкостью. Как же можно выбрать цикл, наиболее подходящий для решения конкретной задачи?

ОТВЕТ. Если количество итераций известно заранее, то лучше выбрать цикл `for`. Цикл `while` оказывается наиболее удобным тогда, когда число повторений цикла заранее неизвестно. В тех случаях, когда требуется, чтобы была выполнена хотя бы одна итерация, используйте цикл `do-while`.

Цикл do-while

Третьей и последней разновидностью циклов в Java является цикл `do-while`. В отличие от циклов `for` и `while`, в которых условие проверялось в самом начале (предусловие), в цикле `do-while` условие выполнения проверяется в самом конце (постусловие). Это означает, что цикл `do-while` всегда выполняется хотя бы один раз. Ниже приведен общий синтаксис цикла `do-while`.

```
do {
   инструкции;
} while (условие);
```

При наличии лишь одной инструкции фигурные скобки в данной форме записи необязательны. Тем не менее они зачастую используются для того, чтобы сделать цикл `do-while` более удобочитаемым и не путать его с циклом `while`. Цикл `do-while` выполняется до тех пор, пока условное выражение истинно.

```
// Демонстрация использования цикла do-while
class DWDemo {
   public static void main(String args[])
      throws java.io.IOException {

      char ch;
```

```
        do {
            System.out.print("Нажмите нужную клавишу, а затем
                              клавишу ENTER: ");
            ch = (char) System.in.read(); // чтение символа с клавиатуры
        } while(ch != 'q');
    }
}
```

Используя цикл `do-while`, мы можем усовершенствовать игру в угадывание букв, созданную в начале главы. На этот раз выполнение цикла будет продолжаться до тех пор, пока пользователь не угадает букву.

```
// Игра в угадывание букв, четвертая версия
class Guess4 {
    public static void main(String args[])
        throws java.io.IOException {

        char ch, ignore, answer = 'K';

        do {
            System.out.println("Задумана буква из диапазона A-Z.");
            System.out.print("Попытайтесь ее угадать: ");

            // Получение символа с клавиатуры
            ch = (char) System.in.read();

            // Отбрасывание всех остальных символов во входном буфере
            do {
                ignore = (char) System.in.read();
            } while(ignore != '\n');

            if(ch == answer) System.out.println("** Правильно! **");
            else {
                System.out.print("...Извините, нужная буква находится ");
                if(ch < answer) System.out.println("ближе к концу алфавита");
                else System.out.println("ближе к началу алфавита");
                System.out.println("Повторите попытку!\n");
            }
        } while(answer != ch);
    }
}
```

Ниже приведен один из возможных вариантов выполнения данной программы в интерактивном режиме.

```
Задумана буква из диапазона A-Z.
Попытайтесь ее угадать: A
...Извините, нужная буква находится ближе к концу алфавита
Повторите попытку!

Задумана буква из диапазона A-Z.
Попытайтесь ее угадать: Z
...Извините, нужная буква находится ближе к началу алфавита
Повторите попытку!
```

```
Задумана буква из диапазона A-Z.
Попытайтесь ее угадать: K
** Правильно! **
```

Обратите внимание на еще одну интересную особенность данной программы: в ней применяются два цикла `do-while`. Первый цикл выполняется до тех пор, пока пользователь не введет правильную букву, а второй цикл приведен ниже и требует дополнительных пояснений.

```
// Отбрасывание всех остальных символов во входном буфере
do {
   ignore = (char) System.in.read();
} while(ignore != '\n');
```

Как пояснялось ранее, консольный ввод буферизуется построчно, т.е. для передачи символов, вводимых с клавиатуры, приходится нажимать клавишу <Enter>, что приводит к формированию последовательности символов перевода строки и возврата каретки. Эти символы сохраняются во входном буфере вместе с символами, введенными с клавиатуры. Кроме того, если ввести с клавиатуры несколько символов подряд, не нажав клавишу <Enter>, то они так и останутся во входном буфере.

В рассматриваемом здесь цикле эти символы отбрасываются до тех пор, пока не достигается конец строки. Если не сделать этого, лишние символы будут передаваться программе в качестве выбранной буквы, что не соответствует действительности. (Для того чтобы убедиться в этом, попробуйте закомментировать внутренний цикл `do-while` в исходном коде программы.) После представления ряда других языковых средств Java в главе 10 будут рассмотрены более совершенные способы обработки консольного ввода. Но применение метода `read()` в данной программе дает элементарное представление о принципе действия системы ввода-вывода в Java. Также в данной программе демонстрируется еще один пример применения циклов в практике программирования на Java.

Упражнение 3.2 Расширение справочной системы

`Help2.java`

В этом упражнении нам предстоит расширить справочную систему Java, созданную в упражнении 3.1. В эту версию программы будут добавлены сведения о синтаксисе циклов `for`, `while` и `do-while`. Кроме того, будет реализована проверка действий пользователя, работающего с меню. Цикл будет повторяться до тех пор, пока пользователь не введет допустимое значение.

1. Скопируйте файл `Help.java` в новый файл `Help2.java`.
2. Измените часть программы, ответственную за отображение вариантов, предлагаемых пользователю на выбор. Реализуйте ее с помощью циклов.

```
public static void main(String args[])
   throws java.io.IOException {

   char choice, ignore;

   do {
      System.out.println("Справка:");
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. for");
      System.out.println("  4. while");
      System.out.println("  5. do-while\n");
      System.out.print("Выберите: ");

      choice = (char) System.in.read();

      do {
         ignore = (char) System.in.read();
      } while(ignore != '\n');
   } while( choice < '1' | choice > '5');
```

Обратите внимание на вложенные циклы do-while, используемые с целью избавиться от нежелательных символов, оставшихся во входном буфере. После внесения приведенных выше изменений программа будет отображать меню в цикле до тех пор, пока пользователь не введет числовое значение в пределах от 1 до 5.

3. Дополните инструкцию switch выводом на экран сведений о синтаксисе циклов for, while и do-while.

```
switch(choice) {
   case '1':
      System.out.println("Инструкция if:\n");
      System.out.println("if(условие) инструкция;");
      System.out.println("else инструкция;");
      break;
   case '2':
      System.out.println("Инструкция switch:\n");
      System.out.println("switch(выражение) {");
      System.out.println("  case константа:");
      System.out.println("    последовательность инструкций");
      System.out.println("    break;");
      System.out.println("  // ...");
      System.out.println("}");
      break;
   case '3':
      System.out.println("Цикл for:\n");
      System.out.print("for(инициализация; условие; итерация)");
      System.out.println("  инструкция;");
      break;
   case '4':
      System.out.println("Цикл while:\n");
      System.out.println("while(условие) инструкция;");
```

```
      break;
   case '5':
      System.out.println("Цикл do-while:\n");
      System.out.println("do {");
      System.out.println("  инструкция;");
      System.out.println("} while (условие);");
      break;
}
```

Обратите внимание на то, что в данном варианте инструкции `switch` отсутствует ветвь `default`. А поскольку цикл отображения меню будет выполняться до тех пор, пока пользователь не введет допустимое значение, необходимость в обработке некорректных значений отпадает.

4. Ниже приведен весь исходный код программы из файла `Help2.java`.

```
/*
   Упражнение 3.2

   Расширенная справочная система, в которой для обработки
   результатов выбора из меню используется цикл do-while.
*/
class Help2 {
   public static void main(String args[])
      throws java.io.IOException {

      char choice, ignore;

      do {
         System.out.println("Справка:");
         System.out.println("  1. if");
         System.out.println("  2. switch");
         System.out.println("  3. for");
         System.out.println("  4. while");
         System.out.println("  5. do-while\n");
         System.out.print("Выберите: ");

         choice = (char) System.in.read();

         do {
            ignore = (char) System.in.read();
         } while(ignore != '\n');
      } while( choice < '1' | choice > '5');

      System.out.println("\n");

      switch(choice) {
         case '1':
            System.out.println("Инструкция if:\n");
            System.out.println("if(условие) инструкция;");
            System.out.println("else инструкция;");
            break;
         case '2':
            System.out.println("Инструкция switch:\n");
```

```
            System.out.println("switch(выражение) {");
            System.out.println("  case константа:");
            System.out.println("    последовательность инструкций");
            System.out.println("    break;");
            System.out.println("  // ...");
            System.out.println("}");
            break;
         case '3':
            System.out.println("Цикл for:\n");
            System.out.print("for(инициализация; условие; итерация)");
            System.out.println("  инструкция;");
            break;
         case '4':
            System.out.println("Цикл while:\n");
            System.out.println("while(условие) инструкция;");
            break;
         case '5':
            System.out.println("Цикл do-while:\n");
            System.out.println("do {");
            System.out.println("  инструкция;");
            System.out.println("} while (условие);");
            break;
      }
    }
  }
```

Применение инструкции break для выхода из цикла

С помощью инструкции break можно организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку условия цикла. Когда в теле цикла встречается инструкция break, цикл завершается, а выполнение программы возобновляется с инструкции, следующей после этого цикла. Рассмотрим следующий краткий пример программы.

```
// Применение инструкции break для выхода из цикла
class BreakDemo {
   public static void main(String args[]) {
      int num;

      num = 100;

      // Выполнение цикла до тех пор, пока квадрат значения
      // переменной i меньше значения переменной num
      for(int i=0; i < num; i++) {
         if(i*i >= num) break; // прекращение выполнения цикла,
                               // если i*i >= 100
         System.out.print(i + " ");
      }
      System.out.println("Цикл завершен.");
   }
}
```

В результате выполнения этой программы будет получен следующий результат:

```
0 1 2 3 4 5 6 7 8 9 Цикл завершен.
```

Как видите, цикл `for` организован для выполнения в пределах значений переменной `num` от 0 до 100. Но, несмотря на это, инструкция `break` прерывает цикл раньше, когда квадрат значения переменной `i` становится больше значения переменной `num`.

Инструкция `break` можно применять в любых циклах, предусмотренных в Java, включая и те, что намеренно организованы бесконечными. В качестве примера ниже приведен простой пример программы, в которой вводимые данные читаются до тех пор, пока пользователь не введет с клавиатуры букву **q**.

```
// Чтение вводимых данных до тех пор,
// пока не будет получена буква q
class Break2 {
   public static void main(String args[])
      throws java.io.IOException {

      char ch;

      for( ; ; ) {  // Бесконечный цикл, завершаемый инструкцией break
         ch = (char) System.in.read();// получение символа
                                      // с клавиатуры
         if(ch == 'q') break;
      }
      System.out.println("Вы нажали клавишу q!");
   }
}
```

Если инструкция `break` применяется посреди набора вложенных циклов, то она прерывает выполнение только самого внутреннего цикла. В качестве примера рассмотрим следующую программу.

```
// Применение инструкции break во вложенных циклах
class Break3 {
   public static void main(String args[]) {

      for(int i=0; i<3; i++) {
         System.out.println("Счетчик внешнего цикла: " + i);
         System.out.print("    Счетчик внутреннего цикла: ");

         int t = 0;
         while(t < 100) {
            if(t == 10) break; // прерывание цикла, если t = 10
            System.out.print(t + " ");
            t++;
         }
         System.out.println();
      }
      System.out.println("Циклы завершены.");
   }
}
```

Выполнение этой программы дает следующий результат.

```
Счетчик внешнего цикла: 0
      Счетчик внутреннего цикла: 0 1 2 3 4 5 6 7 8 9
Счетчик внешнего цикла: 1
      Счетчик внутреннего цикла: 0 1 2 3 4 5 6 7 8 9
Счетчик внешнего цикла: 2
      Счетчик внутреннего цикла: 0 1 2 3 4 5 6 7 8 9
Циклы завершены.
```

Как видите, инструкция `break` из внутреннего цикла вызывает прерывание только этого цикла, а на выполнение внешнего цикла она не оказывает никакого влияния.

В отношении инструкции `break` необходимо также иметь в виду следующее. Во-первых, в теле цикла может быть несколько инструкций `break`, но применять их следует очень аккуратно, поскольку чрезмерное их количество обычно приводит к нарушению нормальной структуры кода. И во-вторых, инструкция `break`, выполняющая выход из инструкции `switch`, оказывает воздействие только на эту инструкцию, но не на охватывающие ее циклы.

Применение инструкции break в качестве оператора goto

Помимо инструкции `switch` и циклов, инструкция `break` может быть использована как “цивилизованный” вариант оператора `goto`. В языке Java оператор `goto` отсутствует, поскольку он обеспечивает возможность произвольного перехода в любую точку программы, что способствует созданию плохо структурированного кода. Программы, в которых часто используется оператор `goto`, обычно сложны для восприятия и поддержки. Но в некоторых случаях оператор `goto` может оказаться полезным. Его удобно, например, использовать для экстренного выхода из многократно вложенных циклов.

Для решения подобных задач в Java применяется расширенная форма инструкции `break`, используя которую можно, например, выйти за пределы одного или нескольких блоков кода. Эти блоки не обязаны быть частью циклов или инструкции `switch`. Более того, можно явно указать точку, с которой должно быть продолжено выполнение программы. Для этой цели в расширенной форме инструкции `break` предусмотрена метка. Как будет показано далее, инструкция `break` позволяет воспользоваться всеми преимуществами оператора `goto` и в то же время избежать сложностей, связанных с его применением.

Ниже приведен общий синтаксис инструкции `break` с меткой.

```
break метка;
```

Как правило, *метка* — это имя, обозначающее блок кода. При выполнении расширенной инструкции `break` управление передается за пределы именованного блока. Инструкция `break` с меткой может содержаться непосредственно в именованном блоке или в одном из блоков, входящих в его состав.

Следовательно, рассматриваемый здесь вариант инструкции break можно использовать для выхода из ряда вложенных блоков. Но это языковое средство не позволяет передать управление в блок, не содержащий инструкцию break.

Для того чтобы присвоить имя блоку, нужно поставить перед ним метку. Именовать можно как независимый блок, так и инструкцию, телом которой является блок кода. Роль метки может выполнять любой допустимый в Java идентификатор с двоеточием. Пометив блок кода, метку можно использовать в качестве адресата инструкции break. Благодаря этому выполнение программы возобновляется с конца именованного блока. Например, в приведенном ниже фрагменте кода используются три вложенных блока.

```
// Применение инструкции break с меткой
class Break4 {
   public static void main(String args[]) {
      int i;

      for(i=1; i<4; i++) {
one:    {
two:       {
three:        {
                 System.out.println("\ni равно " + i);
                 if(i==1) break one;  ◄—— Переход по метке
                 if(i==2) break two;
                 if(i==3) break three;

                 // Эта строка кода никогда не будет достигнута
                 System.out.println("не будет выведено");
              }
              System.out.println("После блока three");
           }
           System.out.println("После блока two");
        }
        System.out.println("После блока one");
      }
      System.out.println("После цикла for");
   }
}
```

В результате выполнения этого фрагмента кода будет получен следующий результат.

```
i равно 1
После блока one

i равно 2
После блока two
После блока one

i равно 3
После блока three
После блока two
После блока one
После цикла for
```

Рассмотрим подробнее приведенный выше фрагмент кода, чтобы лучше понять, каким образом получается именно такой результат его выполнения. Когда значение переменной i равно 1, условие первой инструкции if становится истинным, и управление передается в конец блока с меткой one. В результате выводится сообщение "После блока one". Если значение переменной i равно 2, то успешно завершается проверка условия во второй инструкции if, и выполнение программы продолжается с конца блока с меткой two. В результате выводятся по порядку сообщения "После блока two" и "После блока one". Когда же переменная i принимает значение 3, истинным становится условие в третьей инструкции if, и управление передается в конец блока с меткой three. В этом случае выводятся все три сообщения: два упомянутых выше, а также сообщение "После блока three", а затем еще и сообщение "После цикла for".

Обратимся еще к одному примеру. На этот раз инструкция break будет использована для выхода за пределы нескольких вложенных циклов. Когда во внутреннем цикле выполняется инструкция break, управление передается в конец блока внешнего цикла. Этот блок помечен меткой done. В результате происходит выход из всех трех циклов.

```
// Еще один пример применения инструкции break с меткой
class Break5 {
   public static void main(String args[]) {

done:
      for(int i=0; i<10; i++) {
         for(int j=0; j<10; j++) {
            for(int k=0; k<10; k++) {
               System.out.println(k + " ");
               if(k == 5) break done; // переход по метке done
            }
            System.out.println("После цикла по k"); // не выполнится
         }
         System.out.println("После цикла по j"); // не выполнится
      }
      System.out.println("После цикла по i");
   }
}
```

Ниже приведен результат выполнения данного фрагмента кода.

```
0
1
2
3
4
5
После цикла i
```

Расположение метки имеет большое значение, особенно когда речь идет о циклах. Рассмотрим в качестве примера следующий фрагмент кода.

```
// Расположение метки имеет большое значение
class Break6 {
   public static void main(String args[]) {
      int x=0, y=0;

// Здесь метка располагается перед циклом for
stop1: for(x=0; x < 5; x++) {
         for(y = 0; y < 5; y++) {
             if(y == 2) break stop1;
             System.out.println("x и y: " + x + " " + y);
         }
      }

      System.out.println();

// А тут метка располагается непосредственно перед
// открывающей фигурной скобкой
      for(x=0; x < 5; x++)
stop2: {
         for(y = 0; y < 5; y++) {
            if(y == 2) break stop2;
            System.out.println("x и y: " + x + " " + y);
         }
      }
   }
}
```

Вот результат выполнения данной программы.

```
x и y: 0 0
x и y: 0 1

x и y: 0 0
x и y: 0 1
x и y: 1 0
x и y: 1 1
x и y: 2 0
x и y: 2 1
x и y: 3 0
x и y: 3 1
x и y: 4 0
x и y: 4 1
```

В этой программе наборы вложенных циклов идентичны, за исключением того, что в первом наборе метка находится перед внешним циклом `for`. В таком случае при выполнении инструкции `break` управление передается в конец всего блока цикла `for`, а оставшиеся итерации внешнего цикла пропускаются. Во втором наборе метка находится перед открывающей фигурной скобкой блока кода, определяющего тело внешнего цикла. Поэтому при выполнении инструкции `break stop2` управление передается в конец тела внешнего цикла `for`, и далее выполняется очередной его шаг.

Не следует, однако, забывать, что в инструкции break нельзя использовать метку, не определенную в охватывающем блоке. Например, приведенный ниже фрагмент кода некорректен и не пройдет компиляцию.

```
// Этот фрагмент кода содержит ошибку
class BreakErr {
   public static void main(String args[]) {

      one: for(int i=0; i<3; i++) {
         System.out.print("Проход " + i + ": ");
      }

      for(int j=0; j<100; j++) {
         if(j == 10) break one; // НЕВЕРНО!
         System.out.print(j + " ");
      }
   }
}
```

Блок кода, обозначенный меткой one, не содержит инструкцию break, и поэтому управление не может быть передано этому блоку.

Спросим у эксперта

ВОПРОС. Как уже отмечалось, оператор goto нарушает структуру программы, и поэтому более подходящим вариантом является инструкция break с меткой. Но не нарушает ли структуру программы переход в конец внешнего цикла, т.е. в точку, далеко отстоящую от инструкции break?

ОТВЕТ. Действительно, нарушает. Но если явный переход все-таки необходим, то передача управления в конец блока кода сохраняет хоть какое-то подобие структуры, чего нельзя сказать об операторе goto.

Использование инструкции continue

С помощью инструкции continue можно организовать досрочное завершение шага итерации цикла в обход обычной структуры управления циклом. Инструкция continue осуществляет принудительный переход к следующему шагу цикла, пропуская оставшийся невыполненным код. Таким образом, инструкция continue служит своего рода дополнением к инструкции break. В приведенном ниже примере программы инструкция continue используется в качестве вспомогательного средства для вывода четных чисел в пределах от 0 до 100.

```
// Применение инструкции continue
class ContDemo {
   public static void main(String args[]) {
      int i;
```

```
        // Вывод четных чисел в пределах от 0 до 100
        for(i = 0; i<=100; i++) {
            if((i%2) != 0) continue; // завершение шага итерации цикла
            System.out.println(i);
        }
    }
}
```

В данном примере выводятся только четные числа, поскольку при обнаружении нечетного числа шаг итерации цикла завершается досрочно в обход вызова метода `println()`.

В циклах `while` и `do-while` инструкция `continue` вызывает передачу управления непосредственно условному выражению, после чего продолжается процесс выполнения цикла. А в цикле `for` сначала вычисляется итерационное выражение, затем условное, после чего цикл продолжается.

Как и в инструкции `break`, в инструкции `continue` может быть указана метка, обозначающая тот охватывающий цикл, выполнение которого должно быть продолжено. Ниже приведен пример программы, демонстрирующий применение инструкции `continue` с меткой.

```
// Применение инструкции continue с меткой
class ContToLabel {
    public static void main(String args[]) {

outerloop:
        for(int i=1; i < 10; i++) {
            System.out.print("\nВнешний цикл: проход " + i +
                             ", внутренний цикл: ");
            for(int j = 1; j < 10; j++) {
                if(j == 5) continue outerloop; // продолжение внешнего
                                               // цикла
                System.out.print(j);
            }
        }
    }
}
```

Выполнение этой программы дает следующий результат.

```
Внешний цикл: проход 1, Внутренний цикл: 1234
Внешний цикл: проход 2, Внутренний цикл: 1234
Внешний цикл: проход 3, Внутренний цикл: 1234
Внешний цикл: проход 4, Внутренний цикл: 1234
Внешний цикл: проход 5, Внутренний цикл: 1234
Внешний цикл: проход 6, Внутренний цикл: 1234
Внешний цикл: проход 7, Внутренний цикл: 1234
Внешний цикл: проход 8, Внутренний цикл: 1234
Внешний цикл: проход 9, Внутренний цикл: 1234
```

Как следует из приведенного выше примера, при выполнении инструкции continue управление передается внешнему циклу, и оставшиеся итерации внутреннего цикла пропускаются.

В реальных программах инструкция continue применяется очень редко. И объясняется это, в частности, богатым набором инструкций циклов в Java, удовлетворяющих бо́льшую часть потребностей в написании прикладных программ. Но в особых случаях, когда требуется преждевременное прекращение цикла, инструкция continue позволяет сделать это, не нарушая структуру кода.

Упражнение 3.3 Завершение создания справочной системы Java

Help3.java

В этом упражнении вам предстоит завершить построение справочной системы Java, начатое в предыдущих упражнениях. Данная версия будет дополнена сведениями о синтаксисе инструкций break и continue, а также даст пользователю возможность запрашивать сведения о синтаксисе нескольких инструкций. Эта цель достигается путем добавления внешнего цикла, который выполняется до тех пор, пока пользователь не введет с клавиатуры букву **q** вместо номера пункта меню. Поэтапное описание процесса создания программы приведено ниже.

1. Скопируйте файл Help2.java в новый файл Help3.java.
2. Поместите весь исходный код программы в бесконечный цикл for. Выход из этого цикла будет осуществляться с помощью инструкции break, которая получит управление тогда, когда пользователь введет с клавиатуры букву **q**. А поскольку этот цикл включает в себя весь код, то выход из него означает завершение программы.
3. Измените цикл отображения меню.

```
do {
   System.out.println("Справка:");
   System.out.println("  1. if");
   System.out.println("  2. switch");
   System.out.println("  3. for");
   System.out.println("  4. while");
   System.out.println("  5. do-while");
   System.out.println("  6. break");
   System.out.println("  7. continue\n");
   System.out.print("Выберите (q - выход): ");

   choice = (char) System.in.read();
   do {
      ignore = (char) System.in.read();
   } while(ignore != '\n');
} while( choice < '1' | choice > '7' & choice != 'q');
```

Как видите, теперь цикл включает инструкции break и continue. Кроме того, буква q воспринимается в нем как допустимый вариант выбора.

4. **Дополните инструкцию** switch **вариантами выбора для инструкций** break **и** continue.

```
case '6':
   System.out.println("Инструкция break:\n");
   System.out.println("break; или break метка;");
   break;
case '7':
   System.out.println("Инструкция continue:\n");
   System.out.println("continue; или continue метка;");
   break;
```

5. **Ниже приведен весь исходный код программы, находящейся в файле** Help3.java.

```
/*
   Упражнение 3.3

   Завершенная справочная система по управляющим
   инструкциям Java, обрабатывающая многократные запросы.
*/
class Help3 {
   public static void main(String args[])
      throws java.io.IOException {

      char choice, ignore;

      for(;;) {
         do {
            System.out.println("Справка:");
            System.out.println("  1. if");
            System.out.println("  2. switch");
            System.out.println("  3. for");
            System.out.println("  4. while");
            System.out.println("  5. do-while");
            System.out.println("  6. break");
            System.out.println("  7. continue\n");
            System.out.print("Выберите (q - выход): ");

            choice = (char) System.in.read();

            do {
               ignore = (char) System.in.read();
            } while(ignore != '\n');
         } while( choice < '1' | choice > '7' & choice != 'q');

         if(choice == 'q') break;

         System.out.println("\n");

         switch(choice) {
            case '1':
               System.out.println("Инструкция if:\n");
               System.out.println("if(условие) инструкция;");
```

```
                System.out.println("else инструкция;");
                break;
            case '2':
                System.out.println("Инструкция switch:\n");
                System.out.println("switch(выражение) {");
                System.out.println("  case константа:");
                System.out.println("    последовательность инструкций");
                System.out.println("    break;");
                System.out.println("  // ...");
                System.out.println("}");
                break;
            case '3':
                System.out.println("Цикл for:\n");
                System.out.print("for(init; условие; итерация)");
                System.out.println("  инструкция;");
                break;
            case '4':
                System.out.println("Цикл while:\n");
                System.out.println("while(условие) инструкция;");
                break;
            case '5':
                System.out.println("Цикл do-while:\n");
                System.out.println("do {");
                System.out.println("  инструкция;");
                System.out.println("} while (условие);");
                break;
            case '6':
                System.out.println("Инструкция break:\n");
                System.out.println("break; или break метка;");
                break;
            case '7':
                System.out.println("Инструкция continue:\n");
                System.out.println("continue; или continue метка;");
                break;
        }
        System.out.println();
      }
    }
}
```

6. Ниже приведен один из возможных вариантов выполнения данной программы в диалоговом режиме.

```
Справка:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue

Выберите (q - выход): 1
```

```
Инструкция if:

if(условие) инструкция;
else инструкция;

Справка:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. Continue

Выберите (q - выход): 6

Инструкция break:

break; или break метка;

Справка:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. continue

Выберите (q - выход): q
```

Вложенные циклы

Как следует из предыдущих примеров программ, один цикл может быть вложен в другой. С помощью вложенных циклов решаются самые разные задачи. Поэтому, прежде чем завершить рассмотрение циклов в Java, уделим еще немного внимания вложенным циклам. Ниже приведен пример программы, содержащей вложенные циклы `for`. С помощью этих циклов для каждого числа от 2 до 100 определяется ряд множителей, на которые данное число делится без остатка.

```
/*
   Использование вложенных циклов для нахождения
   делителей чисел от 2 до 100
*/
class FindFac {
   public static void main(String args[]) {

      for(int i=2; i <= 100; i++) {
         System.out.print("Делители " + i + ": ");
         for(int j = 2; j < i; j++)
```

```
            if((i%j) == 0) System.out.print(j + " ");
         System.out.println();
      }
   }
}
```

Ниже приведена часть результата выполнения данной программы.

```
Делители 2:
Делители 3:
Делители 4: 2
Делители 5:
Делители 6: 2 3
Делители 7:
Делители 8: 2 4
Делители 9: 3
Делители 10: 2 5
Делители 11:
Делители 12: 2 3 4 6
Делители 13:
Делители 14: 2 7
Делители 15: 3 5
Делители 16: 2 4 8
Делители 17:
Делители 18: 2 3 6 9
Делители 19:
Делители 20: 2 4 5 10
```

В данной программе переменная `i` из внешнего цикла последовательно принимает значения до 2 до 100. А во внутреннем цикле для каждого числа от 2 до текущего значения переменной `i` выполняется проверка, является ли оно делителем `i`. В качестве упражнения попробуйте сделать данную программу более эффективной. (Подсказка: число итераций во внутреннем цикле можно уменьшить.)

Вопросы и упражнения для самопроверки

1. Напишите программу, которая считывает символы с клавиатуры до тех пор, пока не встретится точка. Предусмотрите в программе счетчик пробелов. Сведения о количестве пробелов должны выводиться в конце программы.
2. Каков общий синтаксис многоступенчатой конструкции `if-else-if`?
3. Допустим, имеется следующий фрагмент кода.

```
if(x < 10)
   if(y > 100) {
      if(!done) x = z;
      else y = z;
   }
else System.out.println("error");// к какой инструкции if относится?
```

С какой из инструкций `if` связана последняя ветвь `else`?

4. Напишите цикл for, в котором перебирались бы значения от 1000 до 0 с шагом 2.
5. Корректен ли следующий фрагмент кода?

```
for(int i = 0; i < num; i++)
   sum += i;

count = i;
```

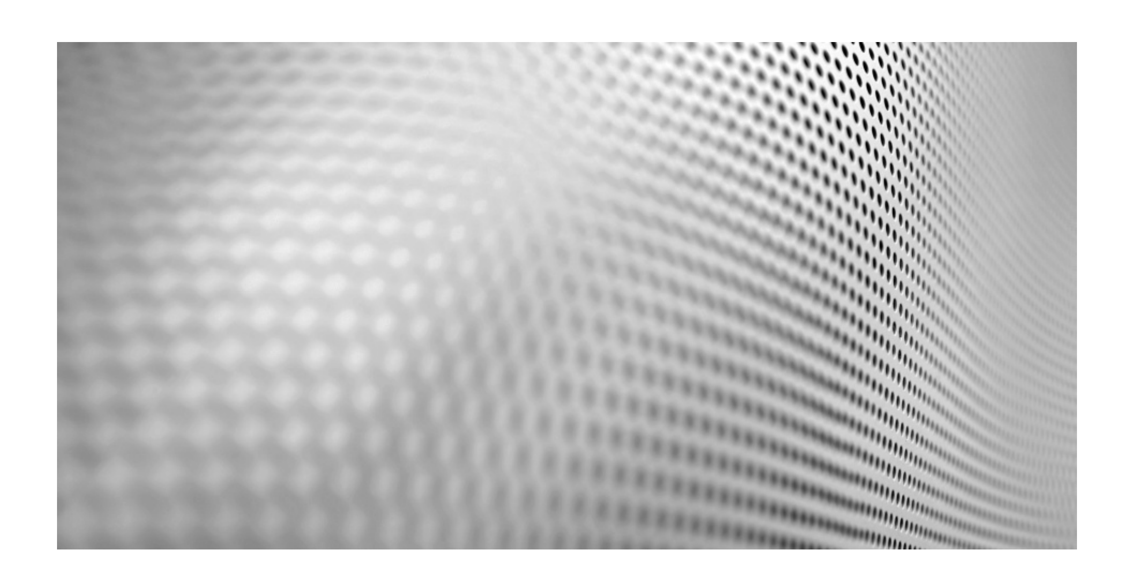
6. Какие действия выполняет инструкция break? Опишите оба варианта этой инструкции.
7. Какое сообщение будет выведено после выполнения инструкции break в приведенном ниже фрагменте кода?

```
for(i = 0; i < 10; i++) {
   while(running) {
      if(x<y) break;
      // ...
   }
   System.out.println("После while");
}
System.out.println("После for");
```

8. Что будет выведено на экран в результате выполнения следующего фрагмента кода?

```
for(int i = 0; i<10; i++) {
   System.out.print(i + " ");
   if((i%2) == 0) continue;
   System.out.println();
}
```

9. Итерационное выражение для цикла for не обязательно должно изменять переменную цикла на фиксированную величину. Эта переменная может принимать произвольные значения. Напишите программу, использующую цикл for для вывода чисел в геометрической прогрессии: 1, 2, 4, 8, 16, 32 и т.д.
10. Коды ASCII-символов нижнего регистра отличается от кодов соответствующих символов верхнего регистра на величину 32. Следовательно, для преобразования строчной буквы в прописную нужно уменьшить ее код на 32. Используйте это обстоятельство для написания программы, читающей символы с клавиатуры. При выводе результатов данная программа должна преобразовывать строчные буквы в прописные, а прописные — в строчные. Остальные символы не должны меняться. Работа программы должна завершаться после того, как пользователь введет с клавиатуры точку. И наконец, сделайте так, чтобы программа отображала количество символов, для которых был изменен регистр.
11. Что такое бесконечный цикл?
12. Должна ли метка, используемая вместе с инструкцией break, быть определена в блоке кода, содержащем эту инструкцию?



Глава 4

Знакомство с классами, объектами и методами

В этой главе...

- Основные сведения о классах
- Создание объектов
- Присваивание ссылок на объекты
- Создание методов, возврат значений и работа с параметрами
- Применение ключевого слова `return`
- Возврат значения из метода
- Добавление параметров в метод
- Применение конструкторов
- Создание параметризированных конструкторов
- Ключевое слово `new`
- Сборка мусора и методы завершения
- Применение ключевого слова `this`

Прежде чем продолжить изучение Java, познакомимся с классами, составляющими саму сущность Java, фундамент, на котором построен этот язык программирования, поскольку класс определяет природу объекта. Следовательно, классы служат прочным основанием для объектно-ориентированного программирования на Java. В классе определяются данные и код, который выполняет действия над этими данными и содержится в методах. Эта глава посвящена классам, объектам и методам, поскольку они играют ведущую роль в Java. Имея представление о классах, объектах и методах, вы сможете создавать более сложные программы и сумеете уяснить те элементы языка Java, которые будут описаны в последующих главах.

Основные сведения о классах

Код любой программы, написанной на Java, находится в пределах класса. Именно поэтому мы начали использовать классы уже с первых примеров программ в книге. Разумеется, мы ограничивались лишь самыми простыми классами и не пользовались большинством их возможностей. Как станет ясно в дальнейшем, классы — это намного более эффективное языковое средство, чем можно было бы предположить, имея о них лишь самое ограниченное представление, полученное после прочтения предыдущих глав.

Начнем рассмотрение классов со знакомства с основными понятиями. *Класс* представляет собой шаблон, на основании которого определяется вид объекта. В нем указываются данные и код, который будет оперировать этими данными. Java использует спецификацию класса для конструирования *объектов*. Объекты — это *экземпляры* классов. Таким образом, класс фактически представляет собой описание, в соответствии с которым должны создаваться объекты. Очень важно, чтобы вы понимали: класс — это логическая абстракция. Физическое представление класса в оперативной памяти возникнет лишь после того, как будет создан объект этого класса.

Следует также иметь в виду, что методы и переменные, составляющие класс, принято называть *членами класса*. Для данных, которые являются членами класса, используется также другое название: *переменные экземпляра*.

Общая форма определения класса

При определении класса вы объявляете его конкретный вид и поведение. Для этого указываются содержащиеся в нем переменные экземпляра и оперирующие ими методы. Простейшие классы могут содержать только код или только данные, однако большинство реальных классов содержит и то и другое.

Класс создается с помощью ключевого слова `class`. Ниже приведен упрощенный общий синтаксис определения класса.

```
class имя_класса {
   // объявление переменных экземпляра
   тип переменная1;
   тип переменная2;
   //...
   тип переменнаяN;

   // объявление методов
   тип метод1(параметры) {
      // тело метода
   }
   тип метод2(параметры) {
      // тело метода
   }
   //...
   тип методN(параметры) {
      // тело метода
   }
}
```

Несмотря на отсутствие соответствующего правила в синтаксисе Java, правильно сконструированный класс должен определять одну и только одну логическую сущность. Например, класс, в котором хранятся имена абонентов и номера их телефонов, обычно не будет содержать сведения о фондовом рынке, среднем уровне осадков, периодичности появления пятен на Солнце или другую не относящуюся к делу информацию. Таким образом, в грамотно

спроектированном классе должна быть сгруппирована логически связанная информация. Если же в один и тот же класс помещается логически несвязанная информация, то структурированность кода быстро нарушается.

Классы, которые применялись в приведенных ранее примерах программ, содержали только один метод: `main()`. Но в представленной выше общей форме описания класса метод `main()` не применяется. Его нужно указывать в классе лишь в том случае, если выполнение программы начинается с данного класса. Кроме того, в некоторых приложениях Java метод `main()` вообще не требуется.

Определение класса

Для того чтобы проиллюстрировать особенности работы с классами, создадим класс, который инкапсулирует сведения о транспортных средствах, например о легковых автомобилях, фургонах и грузовиках. Назовем этот класс `Vehicle`. В нем будут храниться следующие сведения: количество пассажиров, емкость топливного бака и среднее потребление топлива (в милях на галлон).

Ниже приведен первый вариант класса `Vehicle`, в котором определены три переменные экземпляра: `passengers`, `fuelcap` и `mpg`. Обратите внимание на то, что в классе `Vehicle` пока еще отсутствуют методы. Они будут добавлены в последующих разделах, а пока в этом классе содержатся лишь данные.

```
class Vehicle {
   int passengers; // количество пассажиров
   int fuelcap;    // емкость топливного бака
   int mpg;        // потребление топлива в милях на галлон
}
```

Объявление класса создает новый тип данных. В данном случае этот тип называется `Vehicle`. Мы будем использовать это имя для объявления объектов типа `Vehicle`. Как уже упоминалось выше, объявление класса — это всего лишь описание типа данных, и реальный объект при этом не создается. Следовательно, приведенный выше код не приводит к появлению объектов типа `Vehicle`.

Для фактического создания реального объекта `Vehicle` потребуется примерно такой код:

```
Vehicle minivan = new Vehicle(); // создание объекта minivan
                                 // типа Vehicle
```

В результате выполнения этого кода создается объект `minivan`, представляющий собой экземпляр класса `Vehicle`. Иными словами, абстрактная оболочка класса обретает "плоть и кровь". Соответствующие подробности мы обсудим немного позже.

Всякий раз, когда вы создаете экземпляр класса, вы фактически создаете объект, содержащий собственные копии всех переменных экземпляра, определенных в классе. Иными словами, каждый объект типа `Vehicle` будет содержать свои копии переменных `passengers`, `fuelcap` и `mpg`. Для обращения к этим переменным используется так называемая ***точечная нотация***, в

соответствии с которой имя переменной указывается после имени объекта и отделяется от него точкой:

```
объект.член
```

В этой форме объект указывается слева, а член класса — справа от точки. Так, если переменной `fuelcap` объекта `minivan` нужно присвоить значение 16, то это можно сделать следующим образом:

```
minivan.fuelcap = 16;
```

Вообще говоря, точечной нотацией можно пользоваться для обращения как к переменным экземпляра, так и к методам.

Ниже приведен пример программы, в которой используется класс `Vehicle`.

```
/* Программа, в которой используется класс Vehicle.

   Присвойте файлу с исходным кодом имя VehicleDemo.java.
*/
class Vehicle {
   int passengers; // количество пассажиров
   int fuelcap;    // емкость топливного бака
   int mpg;        // потребление топлива в милях на галлон
}

// В этом классе объявляется объект типа Vehicle
class VehicleDemo {
   public static void main(String args[]) {
      Vehicle minivan = new Vehicle();
      int range;

      // Присваивание значений полям в объекте minivan
      minivan.passengers = 7;
      minivan.fuelcap = 16;
      minivan.mpg = 21;  ◄——— Обратите внимание на использование точечной
                              нотации для доступа к переменным экземпляра

      // Расчет дальности поездки с полным баком горючего
      range = minivan.fuelcap * minivan.mpg;
      System.out.println("Мини-фургон может перевезти " +
                         minivan.passengers + " пассажиров
                         на расстояние " + range + " миль");
   }
}
```

Назначим файлу, содержащему приведенный выше код, имя `VehicleDemo.java`, поскольку метод `main()` находится не в классе `Vehicle`, а в классе `VehicleDemo`. После компиляции программы будут созданы два файла с расширением `.class`: один — для класса `Vehicle`, другой — для класса `VehicleDemo`. Компилятор Java автоматически помещает каждый класс в отдельный файл с расширением `.class`. Вовсе не обязательно, чтобы классы `Vehicle` и `VehicleDemo` находились в одном и том же исходном файле, их можно поместить в два отдельных файла: `Vehicle.java` и `VehicleDemo.java`.

Для того чтобы воспользоваться этой программой, следует запустить на выполнение файл VehicleDemo.class. В результате на экран будет выведена следующая информация:

```
Мини-фургон может перевезти 7 пассажиров на расстояние 336 миль
```

Прежде чем переходить к рассмотрению других вопросов, примем к сведению следующий основополагающий принцип: каждый объект содержит собственные копии переменных экземпляра, определенных в его классе. Следовательно, содержимое переменных в одном объекте может отличаться от содержимого тех же самых переменных в другом объекте. Между объектами нет никакой связи, за исключением того, что они относятся к одному и тому же типу. Так, если имеются два объекта типа Vehicle, каждый из них содержит собственную копию переменных passengers, fuelcap и mpg, причем значения одноименных переменных в этих двух объектах могут отличаться. Продемонстрируем это на примере приведенной ниже программы (обратите внимание на то, что класс, содержащий метод main(), на этот раз называется TwoVehicles).

```
// В этой программе создаются два объекта класса Vehicle
class Vehicle {
   int passengers; // количество пассажиров
   int fuelcap;    // емкость топливного бака
   int mpg;        // потребление топлива в милях на галлон }

// В этом классе объявляется объект типа Vehicle
class TwoVehicles {
   public static void main(String args[]) {
      Vehicle minivan = new Vehicle();
      Vehicle sportscar = new Vehicle();

      int range1, range2;

      // Присваивание значений полям объекта minivan
      minivan.passengers = 7;
      minivan.fuelcap = 16;
      minivan.mpg = 21;

      // Присваивание значений полям объекта sportscar
      sportscar.passengers = 2;
      sportscar.fuelcap = 14;
      sportscar.mpg = 12;

      // Расчет дальности поездки с полным баком горючего
      range1 = minivan.fuelcap * minivan.mpg;
      range2 = sportscar.fuelcap * sportscar.mpg;

      System.out.println("Мини-фургон может перевезти " +
                         minivan.passengers + " пассажиров
                         на расстояние " + range1 + " миль.");
      System.out.println("Спортивный автомобиль может перевезти " +
```

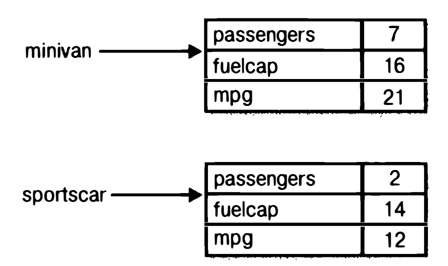
Помните, что переменные minivan и sportscar ссылаются на разные объекты

```
                              sportcar.passengers + " пассажиров на
                              расстояние " + range2 + " миль.");
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Мини-фургон может перевезти 7 пассажиров на расстояние 336 миль.
Спортивный автомобиль может перевезти 2 пассажиров на расстояние
168 миль.
```

Как видите, данные из объекта `minivan` отличаются от соответствующих данных из объекта `sportscar`. Это обстоятельство иллюстрирует приведенный ниже рисунок.



Порядок создания объектов

В рассмотренных ранее примерах программ для объявления объекта типа `Vehicle` использовалась следующая строка кода:

```
Vehicle minivan = new Vehicle();
```

Это объявление выполняет две функции. Во-первых, в нем задается переменная класса `Vehicle` под именем `minivan`. Эта переменная еще не определяет объект, она просто дает возможность *ссылаться на* объект. И во-вторых, в этой строке кода создается физическая копия объекта, а ссылка на него присваивается переменной `minivan`. И делается это с помощью оператора `new`.

Оператор `new` динамически (т.е. во время выполнения программы) выделяет память для объекта и возвращает ссылку на него, которая представляет собой адрес области памяти, выделяемой для объекта оператором `new`. Ссылка на объект сохраняется в переменной. Таким образом, память для объектов всех классов в Java выделяется динамически.

Приведенный выше код можно разбить на две строки, соответствующие отдельным стадиям создания объекта.

```
Vehicle minivan; // объявление ссылки на объект
minivan = new Vehicle(); // выделение памяти для объекта
                         // типа Vehicle
```

В первой строке кода переменная `minivan` объявляется как ссылка на объект типа `Vehicle`. Следует иметь в виду, что `minivan` — это переменная, которая может ссылаться на объект, а не сам объект. В данный момент переменная

minivan пока еще не ссылается на объект. Во второй строке кода создается новый объект типа Vehicle, а ссылка на него присваивается переменной minivan. С этого момента переменная minivan оказывается ассоциированной с объектом.

Переменные ссылочного типа и присваивание

В операции присваивания переменные ссылочного типа ведут себя иначе, чем переменные примитивных типов, например int. Когда одна переменная элементарного типа присваивается другой, ситуация оказывается довольно простой. Переменная, находящаяся в левой части оператора присваивания, получает *копию значения* переменной, находящейся в правой части этого оператора. Если же одна ссылочная переменная присваивается другой, то ситуация несколько усложняется, поскольку такое присваивание приводит к тому, что переменная, находящаяся в левой части оператора присваивания, ссылается на тот же самый объект, что и переменная, находящаяся в правой части этого оператора. Сам же объект не копируется. В силу этого отличия присваивание переменных ссылочного типа может привести к довольно неожиданным результатам. В качестве примера рассмотрим следующий фрагмент кода.

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
```

На первый взгляд кажется, что переменные car1 и car2 ссылаются на совершенно разные объекты, но это не так. Переменные car1 и car2, напротив, ссылаются на один и тот же объект. Когда значение переменной car1 присваивается переменой car2, в конечном итоге переменная car2 ссылается на тот же объект, что и переменная car1. Следовательно, этим объектом можно оперировать с помощью переменной car1 или car2. Например, после очередного присваивания выводится одно и то же значение: 26.

```
car1.mpg = 26;

System.out.println(car1.mpg);
System.out.println(car2.mpg);
```

Несмотря на то что обе переменные, car1 и car2, ссылаются на один и тот же объект, они никак иначе не связаны. Например, в результате приведенной ниже последовательности операций присваивания просто изменяется объект, на который ссылается переменная car2.

```
Vehicle car1 = new Vehicle();
Vehicle car2 = car1;
Vehicle car3 = new Vehicle();

car2 = car3; // теперь переменные car2 и car3
             // ссылаются на один и тот же объект
```

После выполнения этой последовательности операций присваивания переменная car2 ссылается на тот же самый объект, что и переменная car3, а ссылка на объект в переменной car1 не меняется.

Методы

Как отмечалось выше, переменные экземпляра и методы — это две основные составляющие классов. До сих пор класс Vehicle, рассматриваемый здесь в качестве примера, содержал только данные, но не методы. Несмотря на то что классы, включающие лишь одни данные, вполне допустимы, у большинства классов должны быть также методы. Методы — это подпрограммы, которые манипулируют данными, определенными в классе, а во многих случаях предоставляют доступ к этим данным. Как правило, другие части программы взаимодействуют с классом посредством его методов.

Метод состоит из одной или нескольких инструкций. В корректно написанной программе на Java каждый метод выполняет только одну функцию. Каждый метод обладает именем, которое используется для его вызова. Обычно в качестве имени метода можно использовать любой действительный идентификатор. Следует, однако, иметь в виду, что идентификатор main() зарезервирован для метода, с которого начинается выполнение программы. Кроме того, в качестве имен методов нельзя использовать ключевые слова Java.

При упоминании методов в тексте данной книги используется соглашение, ставшее общепринятым в литературе по Java: после названия метода стоит пара круглых скобок. Так, если методу присвоено имя getval, то в тексте книги он упоминается как getval(). Подобная форма записи позволяет отличать имена методов от имен переменных при чтении книги.

Ниже приведен общий синтаксис объявления метода.

```
возращаемый_тип имя(список_параметров) {

    // тело метода

}
```

Здесь *возращаемый_тип* обозначает тип данных, возвращаемых методом. Им может быть любой допустимый тип, в том числе и тип класса, который вы создаете. Если метод не возвращает значение, то для него указывается тип void. Далее, *имя* обозначает конкретное имя, присваиваемое методу. В качестве имени метода может быть использован любой допустимый идентификатор, не приводящий к конфликтам в текущей области видимости. И наконец, *список_параметров* — это последовательность параметров, разделенных запятыми, для каждого из которых указывается тип и имя. Параметры представляют собой переменные, которые получают значения, передаваемые им в виде ***аргументов*** при вызове метода. Если у метода отсутствуют параметры, то список параметров будет пустым.

Добавление метода в класс Vehicle

Как отмечалось ранее, обычно методы класса выполняют действия над данными, входящими в состав класса, и предоставляют доступ к ним. Напомним, что метод main() в предыдущих примерах вычислял дальность поездки транспортного средства. При этом емкость топливного бака умножалась на количество миль, которые может проехать машина, потребив единичный объем топлива (в данном случае — галлон). И хотя такой расчет формально считается правильным, его лучше выполнять в самом классе Vehicle. Преимущества подобного решения очевидны: дальность поездки транспортного средства зависит от потребления топлива в милях на галлон и емкости топливного бака, а обе эти величины инкапсулированы в классе Vehicle. Благодаря добавлению в класс Vehicle метода, предназначенного для расчета искомой величины, улучшается объектно-ориентированная структура кода. Для того чтобы добавить метод в класс Vehicle, его следует объявить в этом классе. Например, приведенный ниже вариант класса Vehicle содержит метод range(), определяющий и отображающий дальность поездки транспортного средства.

```
// Добавление метода range() в класс Vehicle

class Vehicle {
   int passengers; // количество пассажиров
   int fuelcap;    // емкость топливного бака
   int mpg;        // потребление топлива в милях на галлон

   // Отображение дальности поездки транспортного средства
   void range() {   ◄——— Метод range() содержится в классе Vehicle
      System.out.println("Дальность - " + fuelcap * mpg + " миль.");
   }
}
                 Обратите внимание на непосредственное указание переменных fuelcap
                 и mpg без использования точечной нотации
class AddMeth {
   public static void main(String args[]) {
      Vehicle minivan = new Vehicle();
      Vehicle sportscar = new Vehicle();

      int range1, range2;

      // Присваивание значений полям объекта minivan
      minivan.passengers = 7;
      minivan.fuelcap = 16;
      minivan.mpg = 21;

      // Присваивание значений полям объекта sportscar
      sportscar.passengers = 2;
      sportscar.fuelcap = 14;
      sportscar.mpg = 12;
```

```
        System.out.print("Мини-фургон может перевезти " +
                         minivan.passengers + " пассажиров. ");

        minivan.range(); // отображение информации о дальности
                         // поездки мини-фургона

        System.out.print("Спортивный автомобиль может перевезти " +
                         sportscar.passengers + " пассажиров. ");

        sportscar.range(); // отображение дальности поездки
                           // спортивного автомобиля
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Мини-фургон может перевезти 7 пассажиров. Дальность - 336 миль.
Спортивный автомобиль может перевезти 2 пассажиров. Дальность -
168 миль.
```

Рассмотрим основные компоненты данной программы. Начнем с метода `range()`. Первая строка этого метода выглядит так:

```
void range() {
```

В этой строке объявляется метод `range`, для которого не предусмотрены параметры. В качестве типа, возвращаемого этим методом, указано ключевое слово `void`. Таким образом, метод `range()` не возвращает вызывающей части программы никаких данных. И завершается строка открывающей фигурной скобкой, обозначающей начало тела метода.

Тело метода `range()` состоит из единственной строки кода:

```
System.out.println("Дальность поездки - " + fuelcap * mpg + " миль.");
```

которая выводит на экран дальность поездки транспортного средства как результат перемножения значений переменных `fuelcap` и `mpg`. А поскольку у каждого объекта типа `Vehicle` имеются собственные копии переменных `fuelcap` и `mpg`, то при вызове метода `range()` используются данные текущего объекта.

Действие метода `range()` завершается по достижении закрывающей фигурной скобки его тела. После этого управление возвращается вызывающей части программы.

А теперь рассмотрим подробнее следующую строку кода в методе `main()`:

```
minivan.range();
```

в которой вызывается метод `range()` для объекта `minivan`. Чтобы вызвать метод для конкретного объекта, следует указать имя этого объекта перед именем метода, используя точечную нотацию. При вызове метода ему передается управление потоком выполнения программы. Когда метод завершит свое действие, управление будет возвращено вызывающей части программы, и ее выполнение продолжится со строки кода, следующей за вызовом метода.

В данном случае в результате вызова minivan.range() отображается дальность поездки транспортного средства, которое задано объектом minivan. Точно так же при вызове sportscar.range() на экран выводится дальность поездки транспортного средства, которое задано объектом sportscar. При каждом вызове метода range() отображается дальность поездки для указанного объекта.

Необходимо отметить следующую особенность метода range(): обращение к переменным экземпляра fuelcap и mpg осуществляется в нем без применения точечной нотации. Если в методе используется переменная экземпляра, определенная в его классе, то обращаться к ней можно напрямую, не указывая объект. На самом деле такой подход вполне логичен. Ведь метод всегда вызывается относительно некоторого объекта своего класса, а следовательно, при вызове метода объект известен и нет никакой необходимости определять его еще раз. Это означает, что переменные fuelcap и mpg, встречающиеся в теле метода range(), неявно обозначают их копии, находящиеся в том объекте, для которого вызывается метод range().

Возврат из метода

Возврат из метода осуществляется при выполнении одного из двух условий. Первое из них вам уже знакомо по методу range(), а именно: признаком завершения метода и возврата из него служит закрывающая фигурная скобка. Вторым условием является выполнение инструкции return. Существуют две разновидности инструкции return: одна — для методов типа void, не возвращающих значений, а другая — для методов, возвращающих значение вызывающей части программы. Здесь мы рассмотрим первую разновидность инструкции return, а о возвращаемых значениях речь пойдет в следующем разделе.

Реализовать немедленное завершение метода типа void и возврат из него можно с помощью следующей формы инструкции return:

```
return ;
```

При выполнении этой инструкции управление будет возвращено вызывающей части программы, а оставшийся в методе код будет проигнорирован. Рассмотрим в качестве примера следующий метод.

```
void myMeth() {
   int i;

   for(i=0; i<10; i++) {
      if(i == 5) return; // завершение цикла после достижения
                         // значения 5
      System.out.println();
   }
}
```

Здесь переменная цикла for принимает лишь значения от 0 до 5. Как только значение переменной i становится равным 5, цикл завершается, и осуществляется возврат из метода. В одном методе допускается несколько инструкций return. Необходимость в них возникает в том случае, если в методе организовано несколько ветвей выполнения, как в приведенном ниже примере кода.

```
void myMeth() {
   // ...
   if(done) return;
   // ...
   if(error) return;
   // ...
}
```

В данном примере метод возвращает управление вызывающей части программы либо по завершении всех необходимых действий, либо в случае появления ошибки. Применяя инструкции return, следует соблюдать осторожность: слишком большое количество точек возврата из метода нарушает структуру кода. В хорошо спроектированном методе точки выхода из него находятся в хорошо продуманных местах.

Итак, метод с возвращаемым значением типа void может быть завершен одним из двух способов: по достижении закрывающей фигурной скобки тела метода или при выполнении инструкции return.

Возврат значения

Несмотря на то что методы типа void встречаются довольно часто, большинство методов все же возвращают значения. Способность возвращать значение относится к одним из самых полезных свойств метода. Пример возврата значения уже встречался ранее, когда для вычисления квадратного корня использовался метод sqrt().

В программировании возвращаемые значения применяются для самых разных целей. В одних случаях, как, например, при обращении к методу sqrt(), возвращаемое значение представляет собой результат выполнения некоторых вычислений, тогда как в других оно лишь сообщает, успешно ли были выполнены действия, предусмотренные в методе. При этом возвращаемое значение нередко включает код состояния. Независимо от конкретного способа применения, возвращаемые значения являются неотъемлемой частью программирования на Java.

Методы возвращают значения вызывающей части программы, используя следующую форму инструкции return:

```
return значение;
```

где *значение* — конкретное возвращаемое значение. Данная форма инструкции return может быть использована только в тех методах, тип которых отличается от void. Более того, подобные методы обязаны возвращать значение, используя данную форму инструкции return.

Теперь мы можем немного видоизменить метод `range()` с учетом возвращаемых значений. Вместо того чтобы выводить дальность поездки в методе `range()`, лучше ограничиться ее вычислением и возвратом полученного значения. Преимущество такого подхода заключается, в частности, в том, что возвращаемое значение может быть использовано при выполнении других вычислений. Ниже приведен код видоизмененного метода `range()`.

```
// Использование возвращаемого значения
class Vehicle {
   int passengers; // количество пассажиров
   int fuelcap;    // емкость топливного бака
   int mpg;        // потребление топлива в милях на галлон

   // Возврат дальности поездки
   int range() {
      return mpg * fuelcap; ◄——— Возврат дальности поездки для заданного
   }                               транспортного средства
}

class RetMeth {
   public static void main(String args[]) {
      Vehicle minivan = new Vehicle();
      Vehicle sportscar = new Vehicle();

      int range1, range2;

      // Присваивание значений полям объекта minivan
      minivan.passengers = 7;
      minivan.fuelcap = 16;
      minivan.mpg = 21;

      // Присваивание значения полям объекта sportscar
      sportscar.passengers = 2;
      sportscar.fuelcap = 14;
      sportscar.mpg = 12;

      // Получение дальности поездки для разных
      // транспортных средств
      range1 = minivan.range();    ——— Присваивание переменной значения,
      range2 = sportscar.range();      возвращаемого методом

      System.out.println("Мини-фургон может перевезти " +
                         minivan.passengers + " на расстояние " +
                         range1 + " миль.");
      System.out.println("Спортивный автомобиль может перевезти " +
                         sportscar.passengers + " на расстояние " +
                         range2 + " миль.");
   }
}
```

Ниже приведен результат выполнения данной программы.

```
Мини-фургон может перевезти 7 пассажиров на расстояние 336 миль.
Спортивный автомобиль может перевезти 2 пассажиров на расстояние
168 миль.
```

Обратите внимание на то, что вызов метода `range()` в данной программе указывается в правой части оператора присваивания, тогда как в левой части — переменная, которая принимает значение, возвращаемое методом `range()`. Таким образом, после выполнения следующей строки кода значение дальности поездки для объекта `minivan` сохраняется в переменной `range1`:

```
range1 = minivan.range();
```

Следует иметь в виду, что в данном случае метод `range()` возвращает значение типа `int`, т.е. вызывающая часть программы получает целочисленное значение. Тип возвращаемого значения — очень важная характеристика метода, поскольку возвращаемые данные должны соответствовать типу, указанному в определении метода. Иными словами, если метод должен возвращать значение типа `double`, то именно таким и следует объявить его тип.

Несмотря на то что приведенная выше программа компилируется и выполняется без ошибок, ее эффективность можно повысить. В частности, переменные `range1` и `range2` в ней не нужны. Вызов метода `range()` можно непосредственно указать в качестве параметра метода `println()`, как продемонстрировано ниже.

```
System.out.println("Мини-фургон может перевезти " +
                   minivan.passengers + " на расстояние " +
                   minivan.range() + " миль");
```

В данном случае при выполнении метода `println()` будет автоматически осуществляться вызов метод `minivan.range()`, а полученное в итоге значение будет передаваться методу `println()`. Более того, к методу `range()` можно обратиться в любой момент, когда понадобится значение дальности поездки для объекта типа `Vehicle`. В качестве примера ниже приведено выражение, в котором сравнивается дальность поездки двух транспортных средств.

```
if(v1.range() > v2.range()) System.out.println("v1 больше v2");
```

Использование параметров

При вызове метода ему можно передать одно или несколько значений. Значение, передаваемое методу, называется ***аргументом***, тогда как переменная, получающая аргумент, называется ***формальным параметром***, или просто ***параметром***. Параметры объявляются в скобках после имени метода. Синтаксис объявления параметров такой же, как и у переменных. Областью действия параметров является тело метода. За исключением особых случаев передачи аргументов методу параметры действуют так же, как и любые другие переменные.

Ниже приведен простой пример программы, демонстрирующий использование параметров. В классе `ChkNum` метод `isEven()` возвращает логическое значение `true`, если значение, передаваемое при вызове этого метода, является четным числом. В противном случае метод возвращает логическое значение `false`. Таким образом, метод `isEven()` возвращает значение типа `boolean`.

```
// Простой пример применения параметра в методе

class ChkNum {
   // Возврат логического значения true,
   // если x содержит четное число
   boolean isEven(int x) {  ◄——— Здесь x — целочисленный параметр метода isEven()
      if((x%2) == 0) return true;
      else return false;
   }
}

class ParmDemo {
   public static void main(String args[]) {
      ChkNum e = new ChkNum();
                                            Передача аргументов методу isEven()
      if(e.isEven(10)) System.out.println("10 - четное число");

      if(e.isEven(9)) System.out.println("9 - четное число");

      if(e.isEven(8)) System.out.println("8 - четное число");
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
10 - четное число
8 - четное число
```

В данной программе метод `isEven()` вызывается трижды, и каждый раз ему передается новое значение. Рассмотрим подробнее исходный код. Прежде всего обратите внимание на то, каким образом вызывается метод `isEven()`. Его параметр указывается в круглых скобках. При первом вызове методу `isEven()` передается значение 10. Следовательно, когда метод `isEven()` начинает выполняться, параметр `x` получает значение 10. При втором вызове в качестве аргумента этому методу передается значение 9, которое и принимает параметр `x`. А при третьем вызове методу `isEven()` передается значение 8, которое опять же присваивается параметру `x`. Какое бы значение вы ни указали при вызове метода `isEven()`, его все равно получит параметр `x`.

В методе может быть определено несколько параметров, в таком случае они разделяются запятыми. Допустим, в классе `Factor` имеется метод `isFactor()`, который определяет, является ли первый его параметр делителем второго.

```
class Factor {
   boolean isFactor(int a, int b) {  ←—— Этот метод имеет два параметра
      if( (b % a) == 0) return true;
      else return false;
   }
}

class IsFact {
   public static void main(String args[]) {
      Factor x = new Factor();                    Передача двух аргументов
                                                  методу isFactor()
      if(x.isFactor(2, 20)) System.out.println("2 - делитель");
      if(x.isFactor(3, 20)) System.out.println("эта строка не будет
                                               выведена");
   }
}
```

Обратите внимание на то, что при вызове метода `isFactor()` передаваемые ему значения также разделяются запятыми.

При использовании нескольких параметров для каждого из них определяется тип, причем эти типы могут отличаться. Например, следующее объявление метода является корректным.

```
int myMeth(int a, double b, float c) {
   // ...
```

Добавление параметризированного метода в класс Vehicle

Параметризированный метод позволяет реализовать в классе `Vehicle` новую возможность: расчет объема топлива, необходимого для преодоления заданного расстояния. Назовем этот новый метод `fuelneeded()`. Он получает в качестве параметра расстояние в милях, которое должно проехать транспортное средство, а возвращает необходимое для этого количество галлонов топлива. Метод `fuelneeded()` определяется следующим образом.

```
double fuelneeded(int miles) {
   return (double) miles / mpg;
}
```

Обратите внимание на то, что этот метод возвращает значение типа `double`. Это важно, поскольку объем потребляемого топлива не всегда можно выразить целым числом. Ниже приведен исходный код программы для расчета дальности поездки транспортных средств из класса `Vehicle`, включающего метод `fuelneeded()`.

```
/*
   Добавление параметризированного метода, в котором выполняется
   расчет объема топлива, необходимого транспортному средству
   для преодоления заданного расстояния.
*/
```

```
class Vehicle {
   int passengers; // количество пассажиров
   int fuelcap;    // емкость топливного бака
   int mpg;        // потребление топлива в милях на галлон

   // Определение дальности поездки транспортного средства
   int range() {
      return mpg * fuelcap;
   }

   // Расчет количества топлива, необходимого транспортному
   // средству для преодоления заданного расстояния
   double fuelneeded(int miles) {
      return (double) miles / mpg;
   }
}

class CompFuel {
   public static void main(String args[]) {
      Vehicle minivan = new Vehicle();
      Vehicle sportscar = new Vehicle();
      double gallons;
      int dist = 252;

      // Присваивание значений полям объекта minivan
      minivan.passengers = 7;
      minivan.fuelcap = 16;
      minivan.mpg = 21;

      // Присваивание значений полям объекта sportscar
      sportscar.passengers = 2;
      sportscar.fuelcap = 14;
      sportscar.mpg = 12;

      gallons = minivan.fuelneeded(dist);

      System.out.println("Для преодоления " + dist +
                         " миль мини-фургону требуется " +
                         gallons + " галлонов топлива");

      gallons = sportscar.fuelneeded(dist);

      System.out.println("Для преодоления " + dist +
                         " миль спортивному автомобилю требуется " +
                         gallons + " галлонов топлива");
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Для преодоления 252 миль мини-фургону требуется 12.0 галлонов топлива
Для преодоления 252 миль спортивному автомобилю требуется 21.0
галлонов топлива
```

Упражнение 4.1

Создание справочного класса

HelpClassDemo.java

Если попытаться кратко выразить суть понятия *класс*, то потребуется всего одно предложение: класс инкапсулирует функциональные возможности. Иногда трудно определить, где оканчиваются одни функциональные возможности и начинаются другие. Общее правило можно сформулировать так: класс должен служить стандартным блоком для компоновки приложения. Для этой цели класс необходимо спроектировать таким образом, чтобы он представлял собой одну функциональную единицу, выполняющую строго определенные действия. Следовательно, нужно стремиться к тому, чтобы классы были как можно более компактными, но в разумных пределах! Ведь классы, реализующие лишние функциональные возможности, делают код сложным для понимания и плохо структурированным, но классы со слишком ограниченными функциональными возможностями приводят к тому, что программа становится неоправданно фрагментированной. Как же найти золотую середину? В поисках ее наука программирования превращается в *искусство* программирования. Многие программисты считают, что соответствующие навыки приходят с опытом.

В качестве упражнения для приобретения нужных навыков работы с классами вам предстоит преобразовать в класс Help справочную систему, созданную при выполнении упражнения 3.3. Но прежде рассмотрим, какие для этого имеются основания. Во-первых, справочная система представляет собой один логический блок. Эта система отображает лишь синтаксис управляющих инструкций Java. Ее функциональные возможности четко определены. Во-вторых, реализация справочной системы в виде класса представляет собой довольно изящное решение. Всякий раз, когда требуется отобразить подсказку для пользователя, достаточно создать экземпляр объекта справочной системы. И наконец, справочную информацию можно дополнить или изменить, не затрагивая остальные части программы, поскольку она инкапсулирована в классе. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл HelpClassDemo.java. Чтобы сэкономить время и усилия, скопируйте файл Help3.java, созданный вами в процессе работы с упражнением 3.3, и сохраните его под именем HelpClassDemo.java.
2. Для того чтобы преобразовать справочную систему в класс, нужно сначала четко определить ее компоненты. Так, в исходном файле Help3.java программы, реализующей справочную систему, имеется код, отвечающий за отображение меню, получение информации от пользователя, проверку достоверности ответа и отображение данных, соответствующих выбранному пункту меню. В этой программе имеется также цикл, который завершается при вводе символа **q**. По зрелом размышлении становится ясно, что средства организации меню, проверки корректности запроса и отображения

информации являются составными частями справочной системы. В то же время порядок получения данных от пользователя и обработки многократных запросов не имеет к системе непосредственного отношения. Таким образом, нужно создать класс, который отображает справочную информацию, меню для ее выбора и проверяет правильность сделанного выбора. Соответствующие методы класса можно назвать `helpon()`, `showmenu()` и `isvalid()`.

3. Создайте метод `helpon()`, исходный код которого приведен ниже.

```
void helpon(int what) {
   switch(what) {
      case '1':
         System.out.println("Инструкция if:\n");
         System.out.println("if(условие) инструкция;");
         System.out.println("else инструкция;");
         break;
      case '2':
         System.out.println("Инструкция switch:\n");
         System.out.println("switch(выражение) {");
         System.out.println("  case константа:");
         System.out.println("    последовательность инструкций");
         System.out.println("    break;");
         System.out.println("  // ...");
         System.out.println("}");
         break;
      case '3':
         System.out.println("Цикл for:\n");
         System.out.print("for(инициализация; условие; итерация)");
         System.out.println("  инструкция;");
         break;
      case '4':
         System.out.println("Цикл while:\n");
         System.out.println("while(условие) инструкция;");
         break;
      case '5':
         System.out.println("Цикл do-while:\n");
         System.out.println("do {");
         System.out.println("  инструкция;");
         System.out.println("} while (условие;");
         break;
      case '6':
         System.out.println("Инструкция break:\n");
         System.out.println("break; или break метка;");
         break;
      case '7':
         System.out.println("Инструкция continue:\n");
         System.out.println("continue; или continue метка;");
         break;
   }
   System.out.println();
}
```

4. Создайте метод `showmenu()`, исходный код которого приведен ниже.

```
void showmenu() {
   System.out.println("Справка:");
   System.out.println("  1. if");
   System.out.println("  2. switch");
   System.out.println("  3. for");
   System.out.println("  4. while");
   System.out.println("  5. do-while");
   System.out.println("  6. break");
   System.out.println("  7. continue\n");
   System.out.print("Выберите (q - выход): ");
}
```

5. Создайте метод `isvalid()`.

```
boolean isvalid(int ch) {
   if(ch < '1' | ch > '7' & ch != 'q') return false;
   else return true;
}
```

6. Добавьте созданные выше методы в класс `Help`.

```
class Help {
   void helpon(int what) {
      switch(what) {
         case '1':
            System.out.println("Инструкция if:\n");
            System.out.println("if(условие) инструкция;");
            System.out.println("else инструкция;");
            break;
         case '2':
            System.out.println("Инструкция switch:\n");
            System.out.println("switch(выражение) {");
            System.out.println("  case константа:");
            System.out.println("    последовательность инструкций");
            System.out.println("    break;");
            System.out.println("  // ...");
            System.out.println("}");
            break;
         case '3':
            System.out.println("Цикл for:\n");
            System.out.print("for(инициализация; условие; итерация)");
            System.out.println("  инструкция;");
            break;
         case '4':
            System.out.println("Цикл while:\n");
            System.out.println("while(условие) инструкция;");
            break;
         case '5':
            System.out.println("Цикл do-while:\n");
            System.out.println("do {");
            System.out.println("  инструкция;");
            System.out.println("} while (условие;");
            break;
```

```
            case '6':
               System.out.println("Инструкция break:\n");
               System.out.println("break; или break метка;");
               break;
            case '7':
               System.out.println("Инструкция continue:\n");
               System.out.println("continue; или continue метка;");
               break;
         }
         System.out.println();
      }

      void showmenu() {
         System.out.println("Справка:");
         System.out.println("  1. if");
         System.out.println("  2. switch");
         System.out.println("  3. for");
         System.out.println("  4. while");
         System.out.println("  5. do-while");
         System.out.println("  6. break");
         System.out.println("  7. continue\n");
         System.out.print("Выберите (q - выход): ");
      }

      boolean isvalid(int ch) {
         if(ch < '1' | ch > '7' & ch != 'q') return false;
         else return true;
      }
   }
```

7. **Перепишите метод `main()` из упражнения 3.3 таким образом, чтобы использовать в нем новый класс `Help`. Сохраните новый исходный код в файле `HelpClassDemo.java`. Ниже приведен весь исходный код программы, реализующей справочную систему в файле `HelpClassDemo.java`.**

```
/*
   Упражнение 4.1

   Преобразование в класс Help справочной системы
   из упражнения 3.3.
*/

class Help {
   void helpon(int what) {
      switch(what) {
         case '1':
            System.out.println("Инструкция if:\n");
            System.out.println("if(условие) инструкция;");
            System.out.println("else инструкция;");
            break;
         case '2':
            System.out.println("Инструкция switch:\n");
            System.out.println("switch(выражение) {");
```

```
            System.out.println("  case константа:");
            System.out.println("    последовательность инструкций");
            System.out.println("    break;");
            System.out.println("  // ...");
            System.out.println("}");
            break;
         case '3':
            System.out.println("Цикл for:\n");
            System.out.print("for(инициализация; условие; итерация)");
            System.out.println("  инструкция;");
            break;
         case '4':
            System.out.println("Цикл while:\n");
            System.out.println("while(условие) инструкция;");
            break;
         case '5':
            System.out.println("Цикл do-while:\n");
            System.out.println("do {");
            System.out.println("  инструкция;");
            System.out.println("} while (условие;");
            break;
         case '6':
            System.out.println("Инструкция break:\n");
            System.out.println("break; или break метка;");
            break;
         case '7':
            System.out.println("Инструкция continue:\n");
            System.out.println("continue; или continue метка;");
            break;
      }
      System.out.println();
   }

   void showmenu() {
      System.out.println("Справка:");
      System.out.println("  1. if");
      System.out.println("  2. switch");
      System.out.println("  3. for");
      System.out.println("  4. while");
      System.out.println("  5. do-while");
      System.out.println("  6. break");
      System.out.println("  7. continue\n");
      System.out.print("Выберите (q - выход): ");
   }

   boolean isvalid(int ch) {
      if(ch < '1' | ch > '7' & ch != 'q') return false;
      else return true;
   }
}
```

```
class HelpClassDemo {
   public static void main(String args[])
      throws java.io.IOException {
      char choice, ignore;
      Help hlpobj = new Help();

      for(;;) {
         do {
            hlpobj.showmenu();

            choice = (char) System.in.read();

            do {
               ignore = (char) System.in.read();
            } while(ignore != '\n');
         } while( !hlpobj.isvalid(choice) );

         if(choice == 'q') break;

         System.out.println("\n");

         hlpobj.helpon(choice);
      }
   }
}
```

После запуска этой программы вы увидите, что она ведет себя точно так же, как и предыдущая ее версия. Преимущество нынешней версии заключается лишь в том, что теперь справочная система может быть использована повторно всякий раз, когда в этом возникнет потребность.

Конструкторы

В предыдущем примере программы мы вынуждены были вручную устанавливать значения переменных экземпляра для каждого объекта типа `Vehicle`, как показано ниже.

```
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;
```

Но в профессионально написанных Java-программах такой подход вообще не применяется, и на то есть причины. Во-первых, существует большая вероятность допустить ошибку (например, не установить значение одного из полей). И во-вторых, существует гораздо более простой и надежный способ решения подобной задачи: использование конструкторов.

Конструктор инициализирует объект при его создании. Имя конструктора совпадает с именем класса, а с точки зрения синтаксиса он подобен методу. Но у конструкторов нет возвращаемого типа, указываемого явно. Как правило,

конструкторы используются для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта.

Конструкторы имеются у всех классов, независимо от того, определите вы их или нет, поскольку Java автоматически предоставляет конструктор, используемый по умолчанию и инициализирующий все переменные экземпляра их значениями, заданными по умолчанию. Для большинства типов данных значением по умолчанию является нулевое, для типа bool — логическое значение false, а для ссылочных типов — пустое значение null. Но как только вы определите свой собственный конструктор, конструктор по умолчанию предоставляться не будет.

Ниже приведен простой пример, демонстрирующий применение конструктора.

```
// Простой конструктор

class MyClass {
   int x;

   MyClass() {  ◄———————— Конструктор класса MyClass
      x = 10;
   }
}

class ConsDemo {
   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();

      System.out.println(t1.x + " " + t2.x);
   }
}
```

В данном примере конструктор класса MyClass объявляется следующим образом.

```
MyClass() {
   x = 10;
}
```

В этом конструкторе переменной экземпляра x, определяемой в классе MyClass, присваивается значение 10. Этот конструктор вызывается оператором new при создании объекта данного класса. Ниже приведена строка кода, в которой используется оператор new:

```
MyClass t1 = new MyClass();
```

В этой строке кода для объекта t1 вызывается конструктор MyClass(), в котором переменной экземпляра t1.x присваивается значение 10. То же самое

происходит и для объекта `t2`. После вызова данного конструктора переменная экземпляра `t2.x` также получает значение 10. Таким образом, выполнение приведенного выше примера программы дает следующий результат:

```
10 10
```

Параметризированные конструкторы

В предыдущем примере использовался конструктор без параметров. В некоторых случаях этого оказывается достаточно, но зачастую конструктор должен иметь один или несколько параметров. Добавление параметров в конструктор происходит точно так же, как и добавление параметров в метод. Для этого достаточно объявить их в скобках после имени конструктора. Ниже приведен пример применения параметризированного конструктора класса `MyClass`.

```
// Параметризированный конструктор

class MyClass {
   int x;

   MyClass(int i) { ◄——— Этот конструктор имеет параметр
      x = i;
   }
}

class ParmConsDemo {
   public static void main(String args[]) {
      MyClass t1 = new MyClass(10);
      MyClass t2 = new MyClass(88);

      System.out.println(t1.x + " " + t2.x);
   }
}
```

Результат выполнения данной программы выглядит следующим образом:

```
10 88
```

В данной версии программы в конструкторе класса `MyClass` определяется единственный параметр `i`, который используется для инициализации переменной экземпляра `x`. При выполнении следующей строки кода значение 10 сначала передается параметру `i` данного конструктора, а затем присваивается переменной `x`:

```
MyClass t1 = new MyClass(10);
```

Добавление конструктора в класс `Vehicle`

Теперь мы можем усовершенствовать класс `Vehicle`, добавив в него конструктор, в котором будут автоматически инициализироваться поля `passengers`, `fuelcap` и `mpg` при создании объекта. Обратите особое внимание на то, каким образом создаются объекты типа `Vehicle`.

```
// Добавление конструктора

class Vehicle {
   int passengers; // количество пассажиров
   int fuelcap;    // емкость топливного бака
   int mpg;        // потребление топлива в милях на галлон

   // Это конструктор класса Vehicle
   Vehicle(int p, int f, int m) { ◄———— Конструктор класса Vehicle
      passengers = p;
      fuelcap = f;
      mpg = m;
   }

   // Определение дальности поездки транспортного средства
   int range() {
      return mpg * fuelcap;
   }

   // Расчет объема топлива, необходимого транспортному
   // средству для преодоления заданного расстояния
   double fuelneeded(int miles) {
      return (double) miles / mpg;
   }
}

class VehConsDemo {
   public static void main(String args[]) {

      // Завершение создания объектов транспортных средств
      Vehicle minivan = new Vehicle(7, 16, 21);
      Vehicle sportscar = new Vehicle(2, 14, 12);
      double gallons;
      int dist = 252;

      gallons = minivan.fuelneeded(dist);

      System.out.println("Для преодоления " + dist +
                         " миль мини-фургону требуется " +
                         gallons + " галлонов топлива");

      gallons = sportscar.fuelneeded(dist);

      System.out.println("Для преодоления " + dist +
                         " миль спортивному автомобилю требуется " +
                         gallons + " галлонов топлива");
   }
}
```

При создании объекты `minivan` и `sportscar` инициализируются конструктором `Vehicle()`. Каждый такой объект инициализируется параметрами, указанными в конструкторе его класса. Например, в строке кода

```
Vehicle minivan = new Vehicle(7, 16, 21);
```

значения 7, 16 и 21 передаются конструктору `Vehicle()` при создании нового объекта `minivan` с помощью оператора `new`.

В итоге копии переменных `passengers`, `fuelcap` и `mpg` в объекте `minivan` будут содержать значения 7, 16 и 21 соответственно. Рассмотренная здесь версия программы выводит такой же результат, как и ее предыдущая версия.

Еще раз об операторе new

Теперь, когда вы ближе познакомились с классами и их конструкторами, вернемся к оператору `new`, чтобы рассмотреть его более подробно. Вот общий синтаксис этого оператора в контексте присваивания:

```
переменная_класса = new имя_класса(список_аргументов)
```

Здесь *переменная_класса* обозначает имя переменной создаваемого класса, а *имя_класса* — конкретное имя класса, реализуемого в виде экземпляра его объекта. Имя класса и список аргументов в скобках, который может быть пустым, обозначают конструктор этого класса. Если в классе не определен его собственный конструктор, то в операторе `new` будет использован конструктор, предоставляемый в Java по умолчанию. Следовательно, оператор `new` может быть использован для создания объекта, относящегося к классу любого типа. Оператор `new` возвращает ссылку на вновь созданный объект, который получает переменная класса в результате присваивания в данной форме записи.

Оперативная память имеет ограниченный объем, и поэтому вполне возможно, что оператору `new` не удастся выделить память для объекта из-за нехватки доступной памяти. В этом случае генерируется исключение времени выполнения (подробнее об обработке исключений речь пойдет в главе 9). В примерах программ, представленных в книге, ситуация, связанная с исчерпанием оперативной памяти, не учитывается, но при написании реальных программ такую возможность придется принимать во внимание.

Спросим у эксперта

ВОПРОС. Почему оператор `new` не указывается для переменных таких простых типов, как `int` или `float`?

ОТВЕТ. В Java простые типы не реализованы в виде объектов. Ради повышения эффективности кода они реализованы как обычные переменные. Переменная простого типа фактически содержит присвоенное ей значение. Как пояснялось ранее, объектные переменные ссылочного типа представляют собой ссылки на объекты. Такая косвенная адресация (наряду с другими особенностями объектов) вносит дополнительные накладные расходы при работе с объектами. Но подобные издержки не возникают при обращении с простыми типами данных.

Сборка мусора

Как было показано выше, при использовании оператора new память для создаваемых объектов динамически выделяется из пула свободной оперативной памяти. Разумеется, оперативная память не бесконечна, и поэтому свободная память рано или поздно исчерпывается. Это может привести к невозможности выполнения оператора new из-за нехватки памяти, используемой для создания требуемого объекта. Именно по этой причине одной из главных задач любой схемы динамического распределения памяти является своевременное освобождение памяти от неиспользуемых объектов, чтобы сделать ее доступной для последующего перераспределения. Во многих языках программирования освобождение распределенной ранее памяти осуществляется вручную. Например, в С++ для этой цели служит оператор delete. Но в Java применяется другой, более надежный подход: *сборка мусора*.

Подсистема сборки мусора Java освобождает память от лишних объектов автоматически, действуя прозрачно, "за кулисами", и без всякого вмешательства со стороны программиста. Эта подсистема работает следующим образом: если ссылки на объект отсутствуют, то такой объект считается ненужным, и занимаемая им память в итоге возвращается в пул. Эта возвращенная память может быть затем распределена для других объектов.

Сборка мусора осуществляется лишь время от времени по ходу выполнения программы. Она не выполняется сразу же после того, как обнаруживается, что существует один или несколько объектов, которые больше не используются. Обычно, во избежание снижения производительности, сборка мусора выполняется лишь при выполнении двух условий: существуют объекты, подлежащие удалению, и есть необходимость освободить занимаемую ими память. Не забывайте о том, что сборка мусора требует определенных затрат времени, и исполнительная среда Java при выполнении этой операции руководствуется принципом целесообразности. Следовательно, вы никогда не можете знать точно, когда именно произойдет сборка мусора.

Ключевое слово this

И в заключение рассмотрим ключевое слово this. При вызове метода ему автоматически передается ссылка на вызывающий объект, которая обозначается ключевым словом this. Следовательно, ключевое слово this обозначает именно тот объект, по ссылке на который действует вызываемый метод. Поясним назначение ключевого слова this на примере программы, в которой создается класс Pwr, предназначенный для вычисления целочисленной степени заданного числа.

```
class Pwr {
   double b;
   int e;
   double val;
```

```
   Pwr(double base, int exp) {
      b = base;
      e = exp;

      val = 1;
      if(exp==0) return;
      for( ; exp>0; exp--) val = val * base;
   }

   double get_pwr() {
      return val;
   }
}

class DemoPwr {
   public static void main(String args[]) {
      Pwr x = new Pwr(4.0, 2);
      Pwr y = new Pwr(2.5, 1);
      Pwr z = new Pwr(5.7, 0);

      System.out.println(x.b + " в степени " + x.e +
                         " равно " + x.get_pwr());
      System.out.println(y.b + " в степени " + y.e +
                         " равно " + y.get_pwr());
      System.out.println(z.b + " в степени " + z.e +
                         " равно " + z.get_pwr());
   }
}
```

Как вам уже известно, в теле метода можно непосредственно обращаться к другим членам класса, не указывая имя объекта или класса. Так, в методе `get_pwr()` имеется инструкция

```
return val;
```

которая возвращает копию значения переменной `val`, связанной с вызывающим объектом. Эту инструкцию можно переписать в таком виде:

```
return this.val;
```

где ключевое слово `this` ссылается на объект, для которого был вызван метод `get_pwr()`. Следовательно, `this.val` — это ссылка на копию переменной `val` в данном объекте. Таким образом, если бы метод `get_pwr()` был вызван для объекта `x`, ключевое слово `this` в приведенной выше инструкции ссылалось бы на объект `x`. Инструкция, в которой отсутствует ключевое слово `this`, на самом деле является не более чем сокращенной записью.

Ниже приведен исходный код класса `Pwr`, написанный с использованием ключевого слова `this`.

```
class Pwr {
   double b;
   int e;
   double val;
```

```
   Pwr(double base, int exp) {
      this.b = base;
      this.e = exp;

      this.val = 1;
      if(exp==0) return;
      for( ; exp>0; exp--) this.val = this.val * base;
   }

   double get_pwr() {
      return this.val;
   }
}
```

На самом деле ни один программист не напишет класс `Pwr` подобным образом, поскольку добавление ключевого слова `this` не дает никаких преимуществ. В то же время стандартная форма записи тех же инструкций выглядит значительно проще. Но в ряде случаев ключевое слово `this` может оказаться очень полезным. Например, синтаксис языка Java не запрещает использовать имена параметров или локальных переменных, совпадающие с именами глобальных переменных. В таком случае говорят, что локальная переменная или параметр ***скрывает*** переменную экземпляра. При этом доступ к скрытой переменной экземпляра обеспечивается с помощью ключевого слова `this`. Так, приведенный ниже пример конструктора класса `Pwr()` синтаксически правилен, но подобного стиля программирования рекомендуется все же избегать.

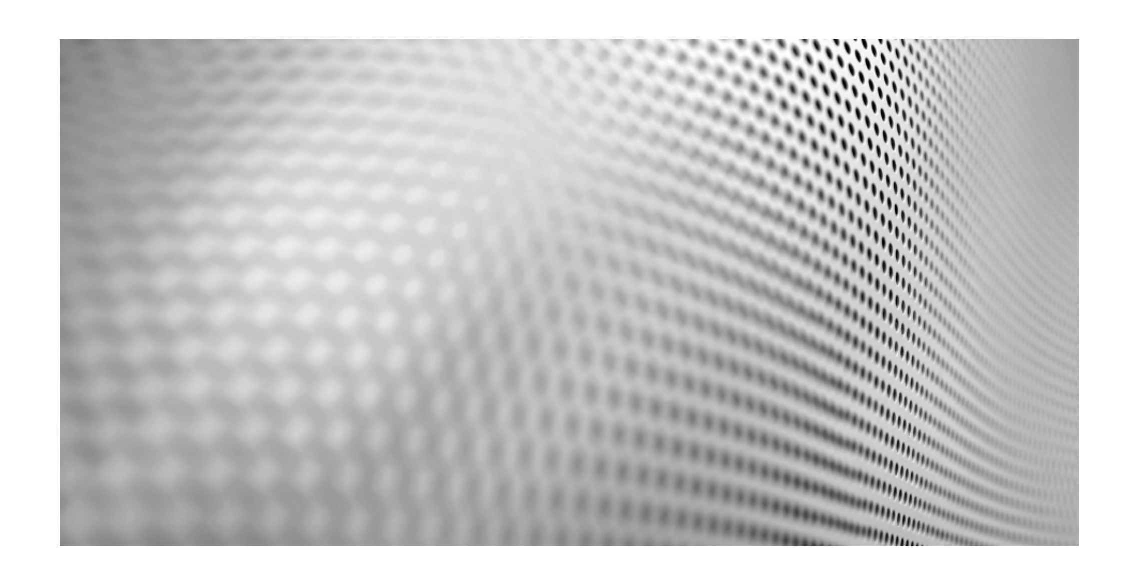
```
Pwr(double b, int e) {
   this.b = b;       // Ссылка на переменную экземпляра, а не на параметр
   this.e = e;

   val = 1;
   if(e==0) return;
   for( ; e>0; e--) val = val * b;
}
```

В данной версии конструктора класса `Pwr` имена параметров совпадают с именами переменных экземпляра, скрывая их. А ключевое слово `this` используется здесь для того, чтобы "открыть" переменные экземпляра.

Вопросы и упражнения для самопроверки

1. Чем отличается класс от объекта?
2. Как определяется класс?
3. Собственную копию чего содержит каждый объект?
4. Покажите, как объявить объект `counter` класса `MyCounter`, используя две отдельные инструкции.

5. Как должен быть объявлен метод myMeth, имеющий два параметра, a и b, типа int и возвращающий значение типа double?
6. Как должно завершаться выполнение метода, возвращающего некоторое значение?
7. Каким должно быть имя конструктора?
8. Какие действия выполняет оператор new?
9. Что такое сборка мусора и для чего она нужна?
10. Что означает ключевое слово this?
11. Может ли конструктор иметь один или несколько параметров?
12. Если метод не возвращает значения, то как следует объявить тип этого метода?



Глава 5

Подробнее о типах данных и операторах

В этой главе...

- Знакомство с массивами
- Создание многомерных массивов
- Создание нерегулярных массивов
- Альтернативный синтаксис объявления массивов
- Присваивание ссылок на массивы
- Применение переменной экземпляра `length` для массивов
- Использование расширенного цикла `for`
- Манипулирование символьными строками
- Использование аргументов командной строки
- Побитовые операторы
- Применение оператора `?`

В этой главе мы возвращаемся к рассмотрению типов данных и операторов Java. В частности, речь пойдет о массивах, классе `String`, побитовых операторах и тернарном операторе `?`. Кроме того, мы рассмотрим разновидность `for-each` цикла `for` и аргументы командной строки.

Массивы

Массив представляет собой совокупность однотипных переменных, объединенных под общим именем. В Java массивы могут быть как одномерными, так и многомерными, хотя чаще всего используются одномерные массивы. Массивы могут применяться для самых разных целей, поскольку они предоставляют удобные средства для объединения связанных вместе переменных. Например, в массиве можно хранить максимальные суточные температуры, зарегистрированные в течение месяца, перечень биржевых курсов или же названия книг по программированию из домашней библиотеки.

Главное преимущество массива — возможность организации данных таким образом, чтобы ими было проще манипулировать. Так, если имеется массив данных о дивидендах, выплачиваемых по избранной группе акций, то, организовав циклическое обращение к элементам этого массива, можно без особого труда рассчитать приносимый этими акциями средний доход. Кроме того, массивы позволяют организовать данные таким образом, чтобы облегчить их сортировку.

Массивами в Java можно пользоваться практически так же, как и в других языках программирования, но у них имеется одна особенность: они реализованы в виде объектов. Именно поэтому их рассмотрение было отложено до тех пор, пока не будут представлены объекты. Реализация массивов в виде объектов дает ряд существенных преимуществ, и далеко не самым последним среди них является возможность освобождения памяти, занимаемой массивами, которые больше не используются, средствами сборки мусора.

Одномерные массивы

Одномерный массив представляет собой список связанных переменных. Такие списки часто применяются в программировании. Например, в одномерном массиве можно хранить номера учетных записей активных пользователей сети или текущие средние показатели игроков бейсбольной команды.

Для объявления одномерного массива обычно применяется общий синтаксис следующего вида:

```
тип имя_массива[] = new тип[размер];
```

Здесь *тип* объявляет конкретный тип массива. Тип массива, называемый также базовым типом, одновременно определяет тип данных каждого элемента, составляющего массив, а *размер* задает число элементов массива. В связи с тем что массивы реализованы в виде объектов, массив создается в два этапа: сначала объявляется переменная, ссылающаяся на массив, а затем выделяется память для массива, и ссылка на нее присваивается переменной массива. Следовательно, память для массивов в Java резервируется динамически с помощью оператора `new`.

Проиллюстрируем вышесказанное на конкретном примере. В следующей строке кода создается массив типа `int`, состоящий из 10 элементов, а ссылка на него присваивается переменной `sample`:

```
int sample[] = new int[10];
```

Объявление массива работает точно так же, как и объявление объекта. В переменной `sample` сохраняется ссылка на область памяти, выделяемую для массива оператором `new`. Этой памяти должно быть достаточно для размещения 10 элементов типа `int`.

Как и объявление объектов, приведенное выше объявление массива можно разбить на два отдельных компонента.

```
int sample[];
sample = new int[10];
```

В данном случае сначала создается переменная `sample`, которая пока что не ссылается на конкретный объект. А затем переменная `sample` получает ссылку на конкретный массив.

Доступ к отдельным элементам массива осуществляется с помощью индексов. *Индекс* обозначает позицию элемента в массиве. В Java индекс первого

элемента массива равен нулю. Так, если массив sample содержит 10 элементов, то их индексы находятся в пределах от 0 до 9. Индексирование массива осуществляется по номерам его элементов, заключенным в квадратные скобки. Например, для доступа к первому элементу массива sample следует указать sample[0], а для доступа к последнему элементу этого массива — sample[9]. В приведенном ниже примере программы в массиве sample сохраняются числа от 0 до 9.

```
// Демонстрация одномерного массива
class ArrayDemo {
   public static void main(String args[]) {
      int sample[] = new int[10];
      int i;

      for(i = 0; i < 10; i = i+1) ◄──┐
         sample[i] = i;              │ Индексация массивов начинается с нуля
                                     │
      for(i = 0; i < 10; i = i+1) ◄──┘
         System.out.println("Элемент[" + i + "]: " + sample[i]);
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
Элемент sample[0]: 0
Элемент sample[1]: 1
Элемент sample[2]: 2
Элемент sample[3]: 3
Элемент sample[4]: 4
Элемент sample[5]: 5
Элемент sample[6]: 6
Элемент sample[7]: 7
Элемент sample[8]: 8
Элемент sample[9]: 9
```

Структура массива sample наглядно показана на следующем рисунке:

0	1	2	3	4	5	6	7	8	9
Sample [0]	Sample [1]	Sample [2]	Sample [3]	Sample [4]	Sample [5]	Sample [6]	Sample [7]	Sample [8]	Sample [9]

Массивы часто используются в программировании, поскольку они позволяют обрабатывать в цикле большое количество переменных. Например, в результате выполнения следующей программы определяются минимальное и максимальное значения из всех, хранящихся в массиве nums. Элементы этого массива последовательно перебираются в цикле for.

```
// Поиск минимального и максимального значений в массиве
class MinMax {
   public static void main(String args[]) {
      int nums[] = new int[10];
      int min, max;

      nums[0] = 99;
      nums[1] = -10;
      nums[2] = 100123;
      nums[3] = 18;
      nums[4] = -978;
      nums[5] = 5623;
      nums[6] = 463;
      nums[7] = -9;
      nums[8] = 287;
      nums[9] = 49;

      min = max = nums[0];
      for(int i=1; i < 10; i++) {
         if(nums[i] < min) min = nums[i];
         if(nums[i] > max) max = nums[i];
      }
      System.out.println("min и max: " + min + " " + max);
   }
}
```

В результате выполнения данной программы будет получен следующий результат:

```
min и max: -978 100123
```

В продемонстрированном выше примере массив nums заполняется вручную в результате выполнения десяти операторов присваивания. И хотя этот способ корректен, можно прибегнуть к более простому методу решения данной задачи. Массивы можно инициализировать в процессе их создания. Для этой цели служит приведенная ниже общая форма инициализации массива.

```
тип имя_массива[] = {val1, val2, val3, ..., valN};
```

Здесь *val1–valN* обозначают начальные значения, которые поочередно присваиваются элементам массива слева направо в направлении увеличения индексов. Java автоматически выделяет объем памяти, достаточный для хранения инициализаторов массивов. При этом необходимость в явном использовании оператора new отпадает сама собой. В качестве примера ниже приведена улучшенная версия программы, в которой определяются максимальное и минимальное значения в массиве.

```
// Применение инициализаторов массива
class MinMax2 {
   public static void main(String args[]) {
      int nums[] = { 99, -10, 100123, 18, -978,
                     5623, 463, -9, 287, 49 };  ◄—— Инициализаторы массива
      int min, max;
```

```
        min = max = nums[0];
        for(int i=1; i < 10; i++) {
            if(nums[i] < min) min = nums[i];
            if(nums[i] > max) max = nums[i];
        }
        System.out.println("min и max: " + min + " " + max);
    }
}
```

Границы массива в Java строго соблюдаются. В случае выхода индекса за верхнюю или нижнюю границу массива при выполнении программы возникает ошибка. Для того чтобы убедиться в этом, попробуйте выполнить приведенную ниже программу, в которой намеренно осуществляется выход за пределы массива.

```
// Намеренный выход за пределы массива
class ArrayErr {
    public static void main(String args[]) {
        int sample[] = new int[10];
        int i;

        // Имитация выхода индекса за пределы массива
        for(i = 0; i < 100; i = i+1)
            sample[i] = i;
    }
}
```

Как только значение переменной `i` достигнет 10, будет сгенерировано исключение `ArrayIndexOutOfBoundsException` и выполнение программы прекратится.

Упражнение 5.1 Сортировка массива

`Bubble.java`

Как отмечалось выше, данные в одномерном массиве организованы в виде индексируемого линейного списка. Такая структура как нельзя лучше подходит для сортировки. В этом упражнении нам предстоит реализовать простой алгоритм сортировки массива. Вам, вероятно, известно, что существуют разные алгоритмы сортировки, в том числе быстрая сортировка, сортировка перемешиванием, сортировка методом Шелла и т.п. Но самым простым и общеизвестным алгоритмом является пузырьковая сортировка. Этот алгоритм не очень эффективен, но отлично подходит для сортировки небольших массивов. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл `Bubble.java`.
2. В алгоритме пузырьковой сортировки соседние элементы массива сравниваются между собой и, если требуется, меняются местами. При этом меньшие значения сдвигаются к одному краю массива, а большие — к другому.

Этот процесс напоминает всплывание пузырьков воздуха на разные уровни в емкости с жидкостью, откуда и произошло название данного алгоритма. Пузырьковая сортировка предполагает обработку массива в несколько проходов. Элементы, взаимное расположение которых отличается от требуемого, меняются местами. Число проходов, необходимых для упорядочения элементов этим способом, на единицу меньше количества элементов в массиве.

Ниже приведен исходный код, составляющий основу алгоритма пузырьковой сортировки. Сортируемый массив называется `nums`.

```
// Пример реализации алгоритма пузырьковой сортировки
for(a=1; a < size; a++)
   for(b=size-1; b >= a; b--) {
      if(nums[b-1] > nums[b]) { // если требуемый порядок следования
                                // не соблюдается, поменять элементы
                                // местами
         t = nums[b-1];
         nums[b-1] = nums[b];
         nums[b] = t;
   }
}
```

Как видите, в приведенном выше фрагменте кода используются два цикла `for`. Во внутреннем цикле сравниваются соседние элементы массива и выявляются элементы, находящиеся не на своих местах. При обнаружении элемента, нарушающего требуемый порядок, два соседних элемента меняются местами. На каждом проходе наименьший элемент перемещается на одну позицию в нужное положение. Внешний цикл обеспечивает повторение описанного алгоритма до завершения всего процесса сортировки.

3. Ниже приведен полный исходный код программы из файла `Bubble.java`.

```
/*
   Упражнение 5.1

   Демонстрация алгоритма пузырьковой сортировки
*/

class Bubble {
   public static void main(String args[]) {
      int nums[] = { 99, -10, 100123, 18, -978,
                     5623, 463, -9, 287, 49 };
      int a, b, t;
      int size;

      size = 10; // количество сортируемых элементов

      // Отображение исходного массива
      System.out.print("Исходный массив:");
      for(int i=0; i < size; i++)
```

```
            System.out.print(" " + nums[i]);
        System.out.println();

        // Реализация алгоритма пузырьковой сортировки
        for(a=1; a < size; a++)
            for(b=size-1; b >= a; b--) {
                if(nums[b-1] > nums[b]) { // если требуемый порядок
                                          // следования не соблюдается,
                                          // поменять элементы местами
                    t = nums[b-1];
                    nums[b-1] = nums[b];
                    nums[b] = t;
                }
            }

        // Отображение отсортированного массива
        System.out.print("Отсортированный массив:");
        for(int i=0; i < size; i++)
            System.out.print(" " + nums[i]);
        System.out.println();
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Исходный массив: 99 -10 100123 18 -978 5623 463 -9 287 49
Отсортированный массив: -978 -10 -9 18 49 99 287 463 5623 100123
```

4. Как упоминалось выше, пузырьковая сортировка отлично подходит для обработки небольших массивов, но при большом числе элементов массива она становится неэффективной. Более универсальным является алгоритм быстрой сортировки, но для его эффективной реализации нужно применять языковые средства Java, которые рассматриваются далее.

Многомерные массивы

Несмотря на то что одномерные массивы применяются чаще всего, в программировании также задействуют и многомерные (двух-, трехмерные и т.д.) массивы. В Java многомерные массивы представляют собой массивы массивов.

Двумерные массивы

Среди многомерных массивов наиболее простыми являются двумерные массивы. Двумерный массив, по сути, представляет собой ряд одномерных массивов. Для того чтобы объявить двумерный целочисленный табличный массив `table` с размерами 10×20, следует использовать следующую строку:

```
int table[][] = new int[10][20];
```

Внимательно изучите эту строку кода, представляющую собой объявление двумерного массива. В отличие от некоторых других языков программирования,

где размеры массива разделяются запятыми, в Java они заключаются в отдельные квадратные скобки. Так, для обращения к элементу массива `table` по индексам 3 и 5 следует указать `table[3][5]`.

В следующем примере двумерный массив заполняется числами от 1 до 12.

```
// Демонстрация использования двумерного массива
class TwoD {
   public static void main(String args[]) {
      int t, i;
      int table[][] = new int[3][4];

      for(t=0; t < 3; ++t) {
         for(i=0; i < 4; ++i) {
            table[t][i] = (t*4)+i+1;
            System.out.print(table[t][i] + " ");
         }
         System.out.println();
      }
   }
}
```

В данном примере кода элемент `table[0][0]` будет содержать значение 1, элемент `table[0][1]` — значение 2, элемент `table[0][2]` — значение 3 и т.д., а элемент `table[2][3]` — значение 12. Структура данного массива представлена в наглядном виде на рис. 5.1.

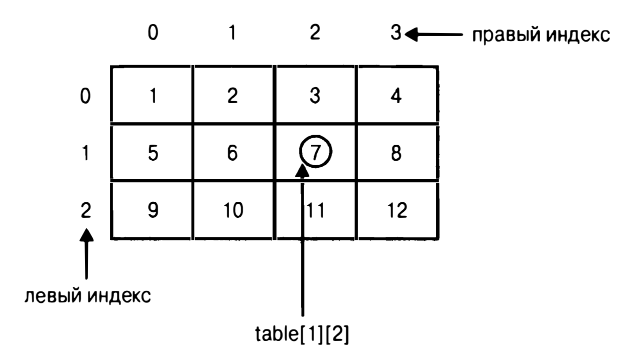


Рис. 5.1. Структура массива `table` из программы `TwoD`

Нерегулярные массивы

При выделении памяти для многомерного массива достаточно указать лишь первый (крайний слева) размер. Память, соответствующую остальным измерениям массива, можно будет выделять отдельно. Например, в приведенном ниже фрагменте кода память выделяется только под первое измерение двумерного массива `table`. Дальнейшее выделение памяти, соответствующей второму измерению, осуществляется вручную.

```
int table[][] = new int[3][];
table[0] = new int[4];
```

```
table[1] = new int[4];
table[2] = new int[4];
```

Объявляя массив подобным способом, мы не получаем никаких преимуществ, но в некоторых случаях подобное объявление оказывается вполне оправданным. В частности, это дает возможность установить разную длину массива по каждому индексу. Как упоминалось выше, многомерный массив реализован в виде массива массивов, что позволяет контролировать размер каждого из них. Допустим, требуется написать программу, в процессе выполнения которой будет сохраняться число пассажиров, перевезенных автобусом-экспрессом в аэропорт. Если автобус делает по десять рейсов в будние дни и по два рейса в субботу и воскресенье, то массив `riders` можно объявить так, как показано в приведенном ниже фрагменте кода. Обратите внимание на то, что длина массива по второму индексу для первых пяти элементов равна 10, а для двух последних — 2.

```
// Выделение памяти по второму индексу массива вручную
class Ragged {
   public static void main(String args[]) {
      int riders[][] = new int[7][];
      riders[0] = new int[10];
      riders[1] = new int[10];
      riders[2] = new int[10];
      riders[3] = new int[10];
      riders[4] = new int[10];
      riders[5] = new int[2];
      riders[6] = new int[2];

      int i, j;

      // Формирование произвольных данных
      for(i=0; i < 5; i++)
         for(j=0; j < 10; j++)
            riders[i][j] = i + j + 10;
      for(i=5; i < 7; i++)
         for(j=0; j < 2; j++)
            riders[i][j] = i + j + 10;

      System.out.println("Количество пассажиров, перевезенных
                          каждым рейсом, в будние дни недели:");
      for(i=0; i < 5; i++) {
         for(j=0; j < 10; j++)
            System.out.print(riders[i][j] + " ");
         System.out.println();
      }
      System.out.println();

      System.out.println("Количество пассажиров, перевезенных
                          каждым рейсом, в выходные дни:");
```

Для первых пяти элементов длина массива по второму индексу равна 10

Для остальных двух элементов длина массива по второму индексу равна 2

```
        for(i=5; i < 7; i++) {
            for(j=0; j < 2; j++)
                System.out.print(riders[i][j] + " ");
            System.out.println();
        }
    }
}
```

Для большинства приложений применение нерегулярных массивов не рекомендуется, поскольку это затрудняет восприятие кода другими программистами. Но в некоторых случаях такие массивы вполне уместны и могут существенно повысить эффективность программ. Так, если вам требуется создать большой двумерный массив, в котором используются не все элементы, нерегулярный массив позволит существенно сэкономить память.

Трехмерные, четырехмерные и многомерные массивы

В Java допускаются массивы размерностью больше двух. Ниже приведена общая форма объявления многомерного массива.

```
тип имя_массива[][]...[] = new тип[размер_1][размер_2]...[размер_N];
```

В качестве примера ниже приведено объявление трехмерного целочисленного массива размером 4×10×3.

```
int multidim[][][] = new int[4][10][3];
```

Инициализация многомерных массивов

Многомерный массив можно инициализировать путем заключения инициализирующей последовательности для каждого размера массива в отдельные фигурные скобки.

```
тип имя_массива[][] = {
    { знач, знач, знач, ..., знач },
    { знач, знач, знач, ..., знач },
.
.
.
    { знач, знач, знач, ..., знач }
};
```

Здесь *знач* обозначает начальное значение, которым инициализируются элементы многомерного массива. Каждый внутренний блок многомерного массива соответствует отдельной строке. В каждой строке первое значение сохраняется в первом элементе подмассива, второе значение — во втором элементе и т.д. Обратите внимание на запятые, разделяющие блоки инициализаторов многомерного массива, а также на точку с запятой после закрывающей фигурной скобки.

В следующем фрагменте кода двумерный массив `sqrs` инициализируется числами от 1 до 10 и их квадратами.

```
// Инициализация двумерного массива
class Squares {
   public static void main(String args[]) {
      int sqrs[][] = {
         { 1, 1 },
         { 2, 4 },
         { 3, 9 },
         { 4, 16 },
         { 5, 25 },
         { 6, 36 },
         { 7, 49 },
         { 8, 64 },
         { 9, 81 },
         { 10, 100 }
      };
      int i, j;

      for(i=0; i < 10; i++) {
        for(j=0; j < 2; j++)
           System.out.print(sqrs[i][j] + " ");
        System.out.println();
      }
   }
}
```

Обратите внимание на то, что у каждой строки свой набор инициализаторов

В результате выполнения этого фрагмента кода будет получен следующий результат.

```
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

Альтернативный синтаксис объявления массивов

Помимо указанной выше общей формы, для объявления массива можно также пользоваться следующим синтаксисом:

```
тип[] имя_переменной;
```

Здесь квадратные скобки указываются после спецификатора типа, а не после имени переменной. Поэтому следующие два объявления равнозначны.

```
int counter[] = new int[3];
int[] counter = new int[3];
```

Равнозначными являются и эти строки кода.

```
char table[][] = new char[3][4];
char[][] table = new char[3][4];
```

Альтернативная форма объявления массива оказывается удобной в тех случаях, когда требуется объявить несколько массивов одного типа. Например:

```
int[] nums, nums2, nums3; // создание трех массивов
```

В этом объявлении создаются три переменные, ссылающиеся на массивы типа `int`. Тот же результат можно получить с помощью следующей строки кода:

```
int nums[], nums2[], nums3[]; // создать три массива
```

Альтернативный синтаксис объявления массива оказывается удобным и в тех случаях, когда в качестве типа, возвращаемого методом, требуется указать массив. Например:

```
int[] someMeth( ) { ...
```

В этой строке кода объявляется метод `someMeth()`, возвращающий целочисленный массив.

Обе рассмотренные выше формы объявления массивов широко распространены в практике программирования на Java и поэтому используются в примерах, представленных в данной книге.

Присваивание ссылок на массивы

Присваивание значения одной переменной, ссылающейся на массив, другой переменной означает, что обе переменные ссылаются на один и тот же массив, и в этом отношении массивы ничем не отличаются от любых других объектов. Подобное присваивание не приводит ни к созданию копии массива, ни к копированию содержимого одного массива в другой. Это продемонстрировано в приведенном ниже примере программы.

```
// Присваивание ссылок на массивы
class AssignARef {
   public static void main(String args[]) {
      int i;

      int nums1[] = new int[10];
      int nums2[] = new int[10];

      for(i=0; i < 10; i++)
         nums1[i] = i;

      for(i=0; i < 10; i++)
         nums2[i] = -i;

      System.out.print("Массив nums1: ");
      for(i=0; i < 10; i++)
         System.out.print(nums1[i] + " ");
      System.out.println();

      System.out.print("Массив nums2: ");
```

```
        for(i=0; i < 10; i++)
           System.out.print(nums2[i] + " ");
        System.out.println();

        nums2 = nums1; // теперь обе переменные ссылаются на
                       // массив nums1

        System.out.print("Массив nums2 после присваивания: ");
        for(i=0; i < 10; i++)
           System.out.print(nums2[i] + " ");
        System.out.println();

        // Выполнение операций над массивом nums1
        // через переменную nums2
        nums2[3] = 99;

        System.out.print("Массив nums1 после изменения через nums2: ");
        for(i=0; i < 10; i++)
           System.out.print(nums1[i] + " ");
        System.out.println();
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Массив nums1: 0 1 2 3 4 5 6 7 8 9
Массив nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Массив nums2 после присваивания: 0 1 2 3 4 5 6 7 8 9
Массив nums1 после изменения через nums2: 0 1 2 99 4 5 6 7 8 9
```

Нетрудно заметить, что в результате присваивания ссылки на массив `nums1` переменной `nums2` обе переменные будут ссылаться на один и тот же массив.

Применение переменной экземпляра length

В связи с тем что массивы реализованы в виде объектов, у каждого массива имеется переменная экземпляра `length`. Значением этой переменной является число элементов, которые может содержать массив. (Иными словами, в переменной `length` содержится информация о размере массива.) Ниже приведен пример программы, демонстрирующий данное свойство массивов.

```
// Демонстрация использования переменной экземпляра length
class LengthDemo {
   public static void main(String args[]) {
      int list[] = new int[10];
      int nums[] = { 1, 2, 3 };
      int table[][] = { // таблица со строками переменной длины
         {1, 2, 3},
         {4, 5},
         {6, 7, 8, 9}
      };
```

```
    System.out.println("Размер list: " + list.length);
    System.out.println("Размер nums: " + nums.length);
    System.out.println("Размер table: " + table.length);
    System.out.println("Размер table[0]: " + table[0].length);
    System.out.println("Размер table[1]: " + table[1].length);
    System.out.println("Размер table[2]: " + table[2].length);
    System.out.println();

    // Использовать переменную length для инициализации списка
    for(int i=0; i < list.length; i++)          ◄─┐
      list[i] = i * i;                            │
                                                  ├── Использование переменной
    System.out.print("Содержимое списка: ");      │   length для управления
    for(int i=0; i < list.length; i++)          ◄─┘   циклом for
       System.out.print(list[i] + " ");
    System.out.println();
  }
}
```

После запуска этой программы будет получен следующий результат.

```
Размер списка: 10
Размер nums: 3
Размер table: 3
Размер table[0]: 3
Размер table[1]: 2
Размер table[2]: 4

Содержимое списка: 0 1 4 9 16 25 36 49 64 81
```

Обратите внимание на то, каким образом переменная `length` используется в двумерном массиве. Как отмечалось ранее, двумерный массив представляет собой массив массивов. Поэтому приведенное ниже выражение позволяет определить число массивов, содержащихся в массиве `table`:

```
table.length
```

Число таких массивов равно 3. Для того чтобы получить размер отдельного массива, содержащегося в массиве `table`, потребуется выражение, аналогичное следующему:

```
table[0].length
```

Это выражение возвращает размер первого массива.

Анализируя код класса `LengthDemo`, следует также отметить, что выражение `list.length` используется в цикле `for` для определения требуемого количества итераций. Учитывая то, что у каждого подмассива своя длина, пользоваться таким выражением удобнее, чем отслеживать вручную размеры массивов. Но не следует забывать, что переменная `length` не имеет никакого отношения к количеству фактически используемых элементов массива. Она содержит лишь данные о том, сколько элементов может включать массив.

Благодаря использованию переменной экземпляра length можно упростить многие алгоритмы. Так, в приведенном ниже примере программы эта переменная используется при копировании одного массива в другой и предотвращает возникновение исключений времени выполнения в связи с выходом за пределы массива.

```
// Пример использования переменной length для копирования массивов
class ACopy {
   public static void main(String args[]) {
      int i;
      int nums1[] = new int[10];
      int nums2[] = new int[10];

      for(i=0; i < nums1.length; i++)
         nums1[i] = i;

      // Копирование массива nums1 в массив nums2
      if(nums2.length >= nums1.length)  ◄—— Использование переменной length
         for(i = 0; i < nums2.length; i++)     для сравнения размеров массивов
            nums2[i] = nums1[i];

      for(i=0; i < nums2.length; i++)
         System.out.print(nums2[i] + " ");
   }
}
```

В данном примере переменная экземпляра length помогает решить две важные задачи. Во-первых, она позволяет убедиться в том, что размера целевого массива достаточно для хранения содержимого исходного массива. И во-вторых, с ее помощью формируется условие завершения цикла, в котором выполняется копирование массива. Конечно, в столь простом примере размеры массивов нетрудно отследить и без переменной экземпляра length, но подобный подход может быть применен для решения более сложных задач.

Упражнение 5.2 Создание класса очереди

QDemo.java

Вам, вероятно, известно, что структура данных — это способ их организации. Одной из самых простых структур является массив, который представляет собой линейный список элементов, допускающий произвольный доступ к ним. Нередко массивы используются в качестве основы для создания более сложных структур наподобие стеков или очередей. *Стек* — это набор элементов с организацией доступа по принципу "первым пришел — последним обслужен", а *очередь* — это набор элементов с организацией доступа по принципу "первым пришел — первым обслужен". Стек можно сравнить со стопкой тарелок на столе: первая тарелка снизу стопки используется последней. А очередь можно сравнить с выстроившейся очередью к кассе супермаркета: покупатель, стоящий в очереди первым, обслуживается первым.

В очередях и стеках нас больше всего интересует способ хранения информации и обращения к ней. И стеки, и очереди представляют собой *механизмы доступа к данным*, в которых хранение и выборка информации поддерживаются самой структурой, а не реализуются в программе. Такое сочетание способов хранения и обработки данных лучше всего реализуется в рамках класса, поэтому в данном упражнении вам предстоит создать простой класс очереди.

В очереди поддерживаются две основные операции: размещение и извлечение. При выполнении операции размещения новый элемент помещается в конец очереди, а при операции извлечения очередной элемент извлекается из начала очереди. Операции с очередью являются *разрушающими*: элемент, однажды извлеченный из очереди, не может быть извлечен из нее повторно. Очередь может быть переполнена, когда в ней не остается места для новых элементов. Но очередь может быть и пуста, когда в ней нет ни одного элемента.

И последнее замечание. Существуют два типа очередей: циклические и нециклические. В *циклической* очереди элементы массива, на основе которого она создана, могут использоваться повторно по мере удаления данных. *Нециклическая* очередь не позволяет повторно использовать элементы, поэтому со временем пространство для хранения новых элементов исчерпывается. Нециклическую очередь создать гораздо проще, чем циклическую, поэтому именно ее мы и реализуем в данном примере для опробования. Но если немного подумать, то нециклическую очередь можно без особого труда превратить в циклическую. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл `QDemo.java`.
2. Очередь можно организовать разными способами. Мы создадим ее на основе массива, выступающего в качестве хранилища данных, помещаемых в очередь. Для доступа к массиву будут использованы два индекса. Индекс *вставки данных* в очередь определяет, в какое место будет помещен следующий элемент очереди. А индекс *извлечения данных* указывает место, откуда должен быть извлечен очередной элемент очереди. Напомним, что операция извлечения является разрушающей и не позволяет извлечь один и тот же элемент дважды. Создаваемая здесь очередь предназначена для хранения символов, но та же самая логика ее организации может быть использована для размещения данных любых типов, в том числе объектов. Итак, начните создание класса очереди `Queue` со следующих строк кода.

```
class Queue {
   char q[]; // массив для хранения элементов очереди
   int putloc, getloc; // индексы для вставки и извлечения
                       // элементов очереди
```

3. Конструктор класса `Queue` создает очередь заданного размера. Его код приведен ниже.

```
Queue(int size) {
   q = new char[size]; // выделение памяти для очереди
   putloc = getloc = 0;
}
```

Обратите внимание на то, что размер очереди на единицу превышает размер, задаваемый параметром `size`. Особенности реализации очереди таковы, что один элемент массива остается неиспользованным, поэтому размер массива должен быть на единицу больше размера очереди, создаваемой на его основе. Первоначально индексы вставки и извлечения данных равны нулю.

4. Метод `put()`, помещающий элемент в очередь, имеет следующий вид.

```
// Помещение символа в очередь
void put(char ch) {
   if(putloc==q.length) {
      System.out.println(" - Очередь заполнена");
      return;
   }

   q[putloc++] = ch;
}
```

Прежде всего в теле данного метода проверяется, не переполнена ли очередь. Если значение переменной `putloc` соответствует последней позиции в массиве `q`, то места для размещения новых элементов в очереди нет. В противном случае переменная `putloc` инкрементируется, и новый элемент располагается в указанном месте массива. Следовательно, переменная `putloc` всегда содержит индекс элемента, помещенного в очередь последним.

5. Для извлечения элементов из очереди используется метод `get()`, код которого приведен ниже.

```
// Извлечение символа из очереди
char get() {
   if(getloc == putloc) {
      System.out.println(" - Очередь пуста");
      return (char) 0;
   }

   return q[getloc++];
}
```

Сначала в данном методе проверяется, пуста ли очередь. Если значения индексов в переменных `getloc` и `putloc` совпадают, то в очереди нет ни одного элемента. Именно поэтому в конструкторе `Queue` переменные `getloc` и `putloc` инициализируются нулевыми значениями. Если очередь не пуста, то переменная `getloc` инкрементируется, и из нее извлекается очередной элемент. Следовательно, переменная `getloc` содержит индекс последнего извлеченного элемента.

6. Ниже приведен полный исходный код программы из файла `QDemo.java`.

```
/*
   Упражнение 5.2

   Класс, реализующий очередь для хранения символов
*/

class Queue {
   char q[];            // массив для хранения элементов очереди
   int putloc, getloc; // индексы для вставки и извлечения
                        // элементов очереди

   Queue(int size) {
      q = new char[size]; // выделение памяти для очереди
      putloc = getloc = 0;
   }

   // Помещение символа в очередь
   void put(char ch) {
      if(putloc==q.length) {
         System.out.println(" - Очередь заполнена");
         return;
      }

      q[putloc++] = ch;
   }

   // Извлечение символа из очереди
   char get() {
      if(getloc == putloc) {
         System.out.println(" - Очередь пуста");
         return (char) 0;
      }

      return q[getloc++];
   }
}

// Демонстрация использования класса Queue
class QDemo {
   public static void main(String args[]) {
      Queue bigQ = new Queue(100);
      Queue smallQ = new Queue(4);
      char ch;
      int i;

      System.out.println("Использование очереди bigQ для
                          сохранения алфавита");
      // Помещение буквенных символов в очередь bigQ
      for(i=0; i < 26; i++)
         bigQ.put((char) ('A' + i));
```

```
        // Извлечение и отображение буквенных символов из
        // очереди bigQ
        System.out.print("Содержимое очереди bigQ: ");
        for(i=0; i < 26; i++) {
           ch = bigQ.get();
           if(ch != (char) 0) System.out.print(ch);
        }

        System.out.println("\n");

        System.out.println("Использование очереди smallQ
                          для генерации ошибок");
        // Использование очереди smallQ для генерации ошибок
        for(i=0; i < 5; i++) {
           System.out.print("Попытка сохранения " + (char) ('Z' - i));

           smallQ.put((char) ('Z' - i));

           System.out.println();
        }
        System.out.println();

        // Дополнительные ошибки при обращении к очереди smallQ
        System.out.print("Содержимое очереди smallQ: ");
        for(i=0; i < 5; i++) {
           ch = smallQ.get();

           if(ch != (char) 0) System.out.print(ch);
        }
    }
}
```

7. Ниже приведен результат выполнения данной программы.

```
Использование очереди bigQ для сохранения алфавита
Содержимое очереди bigQ: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Использование очереди smallQ для генерации ошибок
Попытка сохранения Z
Попытка сохранения Y
Попытка сохранения X
Попытка сохранения W
Попытка сохранения V - Очередь заполнена

Содержимое очереди smallQ: ZYXW - Очередь пуста
```

8. Попытайтесь самостоятельно усовершенствовать класс Queue таким образом, чтобы в очереди можно было хранить другие типы данных, например значения типа int или double.

Цикл типа for-each

При выполнении операций с массивами очень часто возникают ситуации, когда должен быть обработан каждый элемент массива. Например, для расчета суммы всех значений, содержащихся в массиве, нужно обратиться ко всем его элементам. То же самое приходится делать при расчете среднего значения, поиске элемента и решении многих других задач. В связи с тем, что задачи, предполагающие обработку всего массива, встречаются очень часто, в Java была реализована еще одна разновидность цикла `for`, рационализирующая подобные операции с массивами.

Вторая разновидность цикла `for` реализует цикл типа `for-each`, в котором происходит последовательное обращение к каждому элементу совокупности объектов (например, массива). За последние годы циклы `for-each` появились практически во всех языках программирования. Изначально в Java подобный цикл не предусматривался и был реализован лишь в пакете JDK 5. Цикл типа `for-each` называется также ***расширенным циклом*** *`for`*.

Общая форма цикла типа `for-each` выглядит так:

```
for(тип итер_пер : коллекция) блок_инструкций
```

где *тип* обозначает конкретный тип *итер_пер* — итерационной переменной, в которой сохраняются поочередно перебираемые элементы набора данных, обозначенного как *коллекция*. В данной разновидности цикла `for` могут быть использованы разные типы коллекций, но в этой книге рассматриваются только массивы. На каждом шаге цикла очередной элемент извлекается из коллекции и сохраняется в итерационной переменной. Выполнение цикла продолжается до тех пор, пока не будут получены все элементы коллекции. Таким образом, при обработке массива размером *N* в расширенном цикле `for` будут последовательно извлечены элементы с индексами от 0 до *N–1*.

Итерационная переменная получает значения из коллекции, и поэтому ее тип должен совпадать (или, по крайней мере, быть совместимым) с типом элементов, которые содержит коллекция. В частности, при обработке массива тип итерационной переменной должен совпадать с типом массива.

Для того чтобы стали понятнее причины, побудившие к внедрению цикла типа `for-each` в Java, рассмотрим приведенный ниже фрагмент кода, в котором традиционный цикл `for` используется для вычисления суммы значений элементов массива.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

Для того чтобы вычислить упомянутую выше сумму, придется перебрать все элементы массива `nums` от начала до конца. Перебор элементов осуществляется благодаря использованию переменной цикла `i` в качестве индекса массива `nums`. Кроме того, нужно явно указать начальное значение переменной цикла, шаг ее приращения на каждой итерации и условие завершения цикла.

При использовании цикла типа for-each некоторые перечисленные выше действия выполняются автоматически. В частности, отпадает необходимость в использовании переменной цикла, задании ее исходного значения и условия завершения цикла, а также в индексировании массива. Вместо этого массив автоматически обрабатывается в цикле от начала до конца. Код, позволяющий решить ту же самую задачу с помощью цикла типа for-each приведен ниже.

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

На каждом шаге этого цикла переменная x автоматически принимает значение, равное очередному элементу массива nums. Сначала ее значение равно 1, на втором шаге цикла итерации оно становится равным 2 и т.д. В данном случае не только упрощается синтаксис, но и исключается ошибка, связанная с выходом за пределы массива.

Спросим у эксперта

ВОПРОС. Какие типы коллекций, помимо массивов, можно обрабатывать с помощью цикла типа for-each?

ОТВЕТ. Наиболее важное применение цикл типа for-each находит в обработке содержимого коллекций, определенных в Collections Framework — библиотеке классов, реализующих различные структуры данных, в том числе списки, векторы, множества и отображения. Рассмотрение библиотеки Collections Framework выходит за рамки данной книги, а дополнительные сведения о ней можно найти в книге *Java. Полное руководство, 10-е издание.*

Ниже приведен весь исходный код программы, демонстрирующей решение описанной выше задачи с помощью цикла типа for-each.

```
// Использование цикла типа for-each
class ForEach {
   public static void main(String args[]) {
      int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
      int sum = 0;

      // Использование цикла типа for-each для
      // суммирования и отображения значений
      for(int x : nums) {   ←———— Цикл типа for-each
         System.out.println("Значение: " + x);
         sum += x;
      }

      System.out.println("Сумма: " + sum);
   }
}
```

Результат выполнения данной программы выглядит так.

```
Значение: 1
Значение: 2
Значение: 3
Значение: 4
Значение: 5
Значение: 6
Значение: 7
Значение: 8
Значение: 9
Значение: 10
Сумма: 55
```

Нетрудно заметить, что в данной разновидности цикла `for` элементы массива автоматически извлекаются один за другим в порядке возрастания индекса.

Несмотря на то что в расширенном цикле `for` обрабатываются все элементы массива, этот цикл можно завершить преждевременно, используя инструкцию `break`. Так, в цикле, используемом в следующем примере, вычисляется сумма только пяти элементов массива `nums`.

```
// Суммирование первых 5 элементов массива
for(int x : nums) {
   System.out.println("Значение: " + x);
   sum += x;
   if(x == 5) break; // прерывание цикла по достижении значения 5
}
```

Однако следует иметь в виду одну важную особенность цикла типа `for-each`. Итерационная переменная в этом цикле обеспечивает только чтение элементов массива, но ее нельзя использовать для записи значения в какой-либо элемент массива. Иными словами, изменить содержимое массива, присвоив итерационной переменной новое значение, не удастся. Рассмотрим в качестве примера следующую программу.

```
// Циклы типа for-each предназначены только для чтения
class NoChange {
   public static void main(String args[]) {
      int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

      for(int x : nums) {
         System.out.print(x + " ");
         x = x * 10;        ◄——— Эта операция не изменяет содержимое массива nums
      }

      System.out.println();

      for(int x : nums)
         System.out.print(x + " ");

      System.out.println();
   }
}
```

В первом цикле for значение итерационной переменной умножается на 10, но это не оказывает никакого влияния на содержимое массива nums, что и демонстрирует второй цикл for. Это же подтверждает и результат выполнения программы.

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Циклическое обращение к элементам многомерных массивов

Расширенный цикл for используется также при работе с многомерными массивами. Выше уже говорилось, что в Java многомерный массив представляет собой *массив массивов* (например, двумерный массив — это массив, элементами которого являются одномерные массивы). Эту особенность важно помнить, организуя циклическое обращение к многомерным массивам, поскольку на каждом шаге цикла извлекается *очередной массив*, а не отдельный элемент. Более того, итерационная переменная в расширенном цикле for должна иметь тип, совместимый с типом извлекаемого массива. Так, при обращении к двумерному массиву итерационная переменная должна представлять собой ссылку на одномерный массив. При использовании цикла типа for-each для обработки *N*-мерного массива извлекаемый объект представляет собой (*N-1*)-мерный массив. Для того чтобы сказанное стало более понятным, рассмотрим приведенный ниже пример программы, где для извлечения элементов двумерного массива используются вложенные циклы for. Обратите внимание на то, каким образом объявляется переменная x.

```
// Использование расширенного цикла for
// для обработки двумерного массива
class ForEach2 {
   public static void main(String args[]) {
      int sum = 0;
      int nums[][] = new int[3][5];

      // Ввод ряда значений в массив nums
      for(int i = 0; i < 3; i++)
         for(int j=0; j < 5; j++)
            nums[i][j] = (i+1)*(j+1);

      // Использование цикла типа for-each для
      // суммирования и отображения значений.
      for(int x[] : nums) { ←———— Обратите внимание на способ объявления переменной x
         for(int y : x) {
            System.out.println("Значение: " + y);
            sum += y;
         }
      }
      System.out.println("Сумма: " + sum);
   }
}
```

Выполнение этой программы дает следующий результат.

```
Значение: 1
Значение: 2
Значение: 3
Значение: 4
Значение: 5
Значение: 2
Значение: 4
Значение: 6
Значение: 8
Значение: 10
Значение: 3
Значение: 6
Значение: 9
Значение: 12
Значение: 15
Сумма: 90
```

Обратите внимание на следующую строку кода:

```
for(int x[] : nums) {
```

Заметьте, каким образом объявлена переменная `x`. Она представляет собой ссылку на одномерный целочисленный массив. Это очень важно, поскольку на каждом шаге цикла `for` из двумерного массива `nums` извлекается очередной массив, начиная с `nums[0]`. А во внутреннем цикле `for` перебираются элементы полученного массива и отображаются их значения.

Использование расширенного цикла for

Цикл типа `for-each` обеспечивает лишь последовательный перебор элементов от начала до конца массива, поэтому может создаться впечатление, будто такой цикл имеет ограниченное применение. Но это совсем не так. Данный механизм циклического обращения применяется в самых разных алгоритмах. Один из самых характерных тому примеров — организация поиска. В приведенном ниже примере программы расширенный цикл `for` применяется для поиска значения в неотсортированном массиве. Выполнение цикла прерывается, если искомый элемент найден.

```
// Поиск в массиве с использованием расширенного цикла for
class Search {
   public static void main(String args[]) {
      int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
      int val = 5;
      boolean found = false;

      // Использование цикла типа for-each для поиска
      // значения переменной val в массиве nums
      for(int x : nums) {
         if(x == val) {
            found = true;
            break;
         }
      }
```

```
        if(found)
            System.out.println("Значение найдено!");
    }
}
```

В данном случае применение расширенного цикла for вполне оправданно, поскольку найти значение в неотсортированном массиве можно, лишь перебрав все его элементы. (Если бы содержимое массива было предварительно отсортировано, то лучше было бы применить более эффективный алгоритм поиска, например двоичный поиск. В этом случае нам пришлось бы использовать другой цикл.) Расширенным циклом for удобно также пользоваться для расчета среднего значения, нахождения минимального и максимального элементов множества, выявления дубликатов значений и т.п.

Теперь, когда цикл типа for-each описан в достаточной степени, он будет еще не раз использоваться там, где это уместно, в примерах программ, представленных в остальной части книги.

Символьные строки

В повседневной работе каждый программист обязательно сталкивается с объектами типа String. Строковый объект определяет символьную строку и поддерживает операции над ней. Во многих языках программирования символьная строка (или просто строка) — это массив символов, но в Java строки — это объекты.

Возможно, вы и не обратили внимание, но класс String фактически уже использовался в примерах программ, начиная с главы 1. При создании строкового литерала на самом деле генерировался объект типа String. Рассмотрим приведенную ниже инструкцию.

```
System.out.println("В Java строки - это объекты.");
```

Наличие в ней строки "В Java строки - это объекты." автоматически приводит к созданию объекта типа String. Следовательно, класс String незримо присутствовал в предыдущих примерах программ. В последующих разделах будет показано, как этим классом пользоваться явным образом. Следует отметить, что класс String настолько обширен, что мы сможем рассмотреть лишь незначительную его часть. Большую часть функциональных возможностей класса String вам предстоит изучить самостоятельно.

Создание строк

Объекты типа String создаются таким же образом, как и объекты других типов. Для этой цели используется конструктор, как продемонстрировано в следующем примере:

```
String str = new String("Привет!");
```

В данном примере создается объект str типа String, содержащий строку "Привет!". Объект типа String можно создать и на основе другого объекта такого же типа, как показано ниже.

```
String str = new String("Привет!");
String str2 = new String(str);
```

После выполнения этих строк кода объект str2 будет также содержать строку "Привет!".

Ниже представлен еще один способ создания объекта типа String.

```
String str = "Строки Java эффективны.";
```

В данном случае объект str инициализируется последовательностью символов "Строки Java эффективны.".

Создав объект типа String, можете использовать его везде, где допускается строковый литерал (последовательность символов, заключенная в кавычки). Например, объект типа String можно передать в качестве параметра методу println() при его вызове:

```
// Знакомство с классом String
class StringDemo {
   public static void main(String args[]) {
      // Различные способы объявления строк
      String str1 = new String("В Java строки - это объекты.");
      String str2 = "Их можно создавать разными способами.";
      String str3 = new String(str2);

      System.out.println(str1);
      System.out.println(str2);
      System.out.println(str3);
   }
}
```

В результате выполнения этой программы будет получен следующий результат:

```
В Java строки - это объекты.
Их можно создавать разными способами.
Их можно создавать разными способами.
```

Операции над символьными строками

Класс String содержит ряд методов, предназначенных для манипулирования строками. Ниже описаны некоторые из них.

boolean equals(*str*)	Возвращает логическое значение true, если текущая строка содержит ту же последовательность символов, что и параметр *str*
int length()	Возвращает длину строки

Окончание таблицы

char charAt(*index*)	Возвращает символ, занимающий в строке позицию, указываемую параметром *index*
int compareTo(*str*)	Возвращает отрицательное значение, если текущая строка меньше строки *str*, нуль, если эти строки равны, и положительное значение, если текущая строка больше строки *str*
int indexOf(*str*)	Выполняет в текущей строке поиск подстроки, определяемой параметром *str*. Возвращает индекс первого вхождения подстроки *str* или -1, если поиск завершается неудачно
int lastIndexOf(*str*)	Выполняет в текущей строке поиск подстроки, определяемой параметром *str*. Возвращает индекс последнего вхождения подстроки *str* или -1, если поиск завершается неудачно

В приведенном ниже примере программы демонстрируется применение перечисленных выше методов работы со строками.

```
// Некоторые операции над строками
class StrOps {
   public static void main(String args[]) {
      String str1 = "Java - лидер Интернета!";
      String str2 = new String(str1);
      String str3 = "Строки Java эффективны.";
      int result, idx;
      char ch;

      System.out.println("Длина str1: " + str1.length());

      // Отображение строки str1 посимвольно
      for(int i=0; i < str1.length(); i++)
         System.out.print(str1.charAt(i));
      System.out.println();

      if(str1.equals(str2))
         System.out.println("str1 эквивалентна str2");
      else
         System.out.println("str1 не эквивалентна str2");

      if(str1.equals(str3))
         System.out.println("str1 эквивалентна str3");
      else
         System.out.println("str1 не эквивалентна str3");

      result = str1.compareTo(str3);
      if(result == 0)
         System.out.println("str1 и str3 равны");
      else if(result < 0)
```

```
         System.out.println("str1 меньше str3");
      else
         System.out.println("str1 больше str3");

      // Присваивание переменной str2 новой строки
      str2 = "One Two Three One";

      idx = str2.indexOf("One");
      System.out.println("Индекс первого вхождения One: " + idx);
      idx = str2.lastIndexOf("One");
      System.out.println("Индекс последнего вхождения One: " + idx);
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Длина str1: 45
Java - лидер Интернета!
str1 эквивалентна str2
str1 не эквивалентна str3
str1 больше str3
Индекс первого вхождения One: 0
Индекс последнего вхождения One: 14
```

Конкатенация — это операция, позволяющая объединить две строки. В коде она обозначается знаком “плюс” (+). Например, в приведенном ниже коде переменная `str4` инициализируется строкой “OneTwoThree”.

```
String str1 = "One";
String str2 = "Two";
String str3 = "Three";
String str4 = str1 + str2 + str3;
```

Спросим у эксперта

ВОПРОС. Для чего в классе `String` определен метод `equals()`? Не проще ли использовать вместо него оператор ==?

ОТВЕТ. Метод `equals()` сравнивает последовательности символов, содержащиеся в двух объектах типа `String`, и проверяет, совпадают ли они, тогда как оператор == позволяет лишь определить, указывают ли две ссылки типа `String` на один и тот же объект.

Массивы строк

Подобно другим типам данных, строки можно объединять в массивы. Ниже приведен соответствующий демонстрационный пример.

```
// Демонстрация использования массивов строк
class StringArrays {
```

```
   public static void main(String args[]) {
      String strs[] = { "Эта", "строка", "является", "тестом." };

      System.out.println("Исходный массив: ");
      for(String s : strs)
         System.out.print(s + " ");
      System.out.println("\n");

      // Изменение строки
      strs[2] = "также является";
      strs[3] = "тестом!";

      System.out.println("Измененный массив: ");
      for(String s : strs)
         System.out.print(s + " ");
   }
}
```

В результате выполнения этого фрагмента кода будет получен следующий результат.

```
Исходный массив:
Эта строка является тестом.

Измененный массив:
Эта строка также является тестом!
```

Неизменяемость строк

Объекты типа `String` неизменяемые. Это означает, что состояние такого объекта не может быть изменено после его создания. Такое ограничение способствует наиболее эффективной реализации строк. Поэтому очевидный, на первый взгляд, недостаток на самом деле превращается в преимущество. Так, если требуется видоизменить уже существующую строку, то следует создать новую строку, содержащую все необходимые изменения. А поскольку неиспользуемые строковые объекты автоматически удаляются сборщиком мусора, то о дальнейшей судьбе ненужных строк можно не беспокоиться. Следует отметить, что содержимое ссылочных переменных типа `String` может изменяться и привести к тому, что переменная будет ссылаться на другой объект, но содержимое самих объектов типа `String` остается неизменным после того, как они были созданы.

Спросим у эксперта

ВОПРОС. Как пояснялось выше, содержимое однажды созданного объекта типа `String` не может быть изменено после его создания. С практической точки зрения это не является серьезным ограничением, но что если мне нужно создать строку, которая может изменяться?

ОТВЕТ. Можете считать, что вам повезло. В Java имеется класс `StringBuffer`, который создает символьные строки, способные изменяться. Так, в дополнение к методу `charAt()`, возвращающему символ из указанного места в строке, в классе `StringBuffer` определен метод `setCharAt()`, включающий символ в строку. Но для большинства целей вполне подходит класс `String`, так что особой необходимости в использовании класса `StringBuffer` не возникает.

Для того чтобы стало понятнее, почему неизменяемость строк не является помехой, воспользуемся еще одним способом обработки строк класса `String` — методом `substring()`, возвращающим новую строку, которая содержит часть вызывающей строки. В итоге создается новый строковый объект, содержащий выбранную подстроку, тогда как исходная строка не меняется, а следовательно, соблюдается принцип постоянства строк. Так выглядит общий синтаксис объявления метода `substring()`:

```
string substring(int начальный_индекс, int конечный_индекс)
```

Здесь *начальный_индекс* обозначает начало извлекаемой подстроки, а *конечный_индекс* — ее окончание. Ниже приведен пример программы, демонстрирующий применение метода `substring()` и принцип неизменяемости строк.

```
// Применение метода substring()
class SubStr {
   public static void main(String args[]) {
      String orgstr = "Java - двигатель Интернета.";

      // Сформировать подстроку
      String substr = orgstr.substring(7, 25); ◄── Здесь создается новая строка, содержащая нужную подстроку

      System.out.println("orgstr: " + orgstr);
      System.out.println("substr: " + substr);
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
orgstr: Java - двигатель Интернета.
substr: двигатель Интернета
```

Как видите, исходная строка `orgstr` остается неизменной, а новая строка `substr` содержит сформированную подстроку.

Использование строк для управления инструкцией `switch`

Как отмечалось в главе 3, до появления версии JDK 7 для управления инструкцией `switch` приходилось использовать лишь константы целочисленных

типов, таких как int или char. Это препятствовало применению инструкции switch в тех случаях, когда выбор варианта определялся содержимым строки. В качестве выхода из этого положения зачастую приходилось обращаться к многоступенчатой конструкции if-else-if. И хотя эта конструкция семантически корректна, для подобного выбора более естественным было бы применение инструкции switch. К счастью, этот недостаток был устранен. После появления комплекта JDK 7 появилась возможность управлять инструкцией switch с помощью объектов типа String. Во многих ситуациях это способствует созданию более удобочитаемого и рационально организованного кода.

Ниже приведен пример программы, демонстрирующий управление инструкцией switch с помощью объектов типа String.

```
// Использование строк для управления инструкцией switch
class StringSwitch {
   public static void main(String args[]) {

      String command = "cancel";

      switch(command) {
         case "connect":
            System.out.println("Подключение");
            break;
         case "cancel":
            System.out.println("Отмена");
            break;
         case "disconnect":
            System.out.println("Отключение");
            break;
         default:
            System.out.println("Неверная команда!");
            break;
      }
   }
}
```

Как и следовало ожидать, выполнение этой программы приводит к следующему результату:

```
Отмена
```

Строка, содержащаяся в переменной command, а в данном примере это "cancel" (Отмена), проверяется на совпадение со строковыми константами в ветвях case инструкции switch. Если совпадение обнаружено, как это имеет место во второй ветви case, выполняется код, связанный с данным вариантом выбора.

Возможность использования строк в инструкции switch очень удобна и позволяет сделать код более удобочитаемым. В частности, применение инструкции switch, управляемой строками, является лучшим решением по сравнению с эквивалентной последовательностью инструкций if-else. Но если учитывать

накладные расходы, то использование строк для управления переключателями оказывается менее эффективным по сравнению с целочисленными значениями. Поэтому использовать строки для данной цели целесообразно лишь в тех случаях, когда управляющие данные уже являются строками. Иными словами, пользоваться строками в инструкции `switch` без особой надобности не следует.

Использование аргументов командной строки

Теперь, когда вы уже познакомились с классом `String`, можно пояснить назначение параметра `args` метода `main()` в исходном коде большинства рассмотренных ранее примеров программ. Многие программы получают параметры, задаваемые в командной строке. Это так называемые ***аргументы командной строки***. Они представляют собой данные, указываемые непосредственно после имени запускаемой программы. Для того чтобы получить доступ к аргументам командной строки из программы на Java, достаточно обратиться к массиву объектов типа `String`, который передается методу `main()`. Рассмотрим в качестве примера программу, отображающую параметры командной строки. Ее исходный код приведен ниже.

```
// Отображение всех данных, указываемых в командной строке
class CLDemo {
   public static void main(String args[]) {
      System.out.println("Программе передано " + args.length +
                         " аргумента командной строки.");

      System.out.println("Список аргументов: ");
      for(int i=0; i<args.length; i++)
         System.out.println("arg[" + i + "]: " + args[i]);
   }
}
```

Допустим, программа `CLDemo` была запущена из командной строки следующим образом:

```
java CLDemo one two three
```

Тогда результат ее выполнения будет следующим.

```
Программе передано 3 аргумента командной строки.
Список аргументов:
arg[0]: one
arg[1]: two
arg[2]: three
```

Обратите внимание на то, что первый аргумент содержится в строке, хранящейся в элементе массива с индексом 0. Для доступа ко второму аргументу следует воспользоваться индексом 1 и т.п.

Для того чтобы стало понятнее, как пользоваться аргументами командной строки, рассмотрим следующую программу, которая получает один аргумент, определяющий имя абонента, а затем выполняет поиск имени в двумерном

массиве строк. Если имя найдено, программа отображает телефонный номер указанного абонента.

```
// Простейший автоматизированный телефонный справочник
class Phone {
   public static void main(String args[]) {
      String numbers[][] = {
         { "Tom", "555-3322" },
         { "Mary", "555-8976" },
         { "John", "555-1037" },
         { "Rachel", "555-1400" }
      };
      int i;

      // Для того чтобы воспользоваться программой,
      // ей нужно передать один аргумент командной строки
      if(args.length != 1)  ◄── Для выполнения программы нужен как минимум один аргумент командной строки
         System.out.println("Использование: java Phone <имя>");
      else {
         for(i=0; i<numbers.length; i++) {
            if(numbers[i][0].equals(args[0])) {
               System.out.println(numbers[i][0] + ": " +
                                  numbers[i][1]);
               break;
            }
         }
         if(i == numbers.length)
            System.out.println("Имя не найдено.");
      }
   }
}
```

Выполнение этой программы может дать, например, следующий результат.

```
C>java Phone Mary
Mary: 555-8976
```

Побитовые операторы

В главе 2 были рассмотрены арифметические и логические операторы, а также операторы сравнения. Эти три вида операторов используются наиболее часто, но в Java предусмотрены также побитовые операторы, существенно расширяющие возможности языка. В побитовых операторах в качестве операндов могут выступать только значения типа `long`, `int`, `short`, `char` и `byte`. К типам `boolean`, `float`, `double` и к классам побитовые операторы неприменимы. Эти операторы называются побитовыми, поскольку они в основном используются для проверки, установки и сдвига отдельных разрядов числа. Побитовые операторы играют чрезвычайно важную роль в системных программах, предназначенных для управления обменом данными с устройствами. Перечень доступных в Java побитовых операторов приведен в табл. 5.1.

Таблица 5.1. Побитовые операторы

Оператор	Операция
&	Побитовое И
|	Побитовое ИЛИ
^	Побитовое исключающее ИЛИ
>>	Сдвиг вправо
>>>	Сдвиг вправо без знака
<<	Сдвиг влево
~	Дополнение до 1 (унарная операция НЕ)

Побитовые операции И, ИЛИ, исключающее ИЛИ и НЕ

Побитовые операторы И (&), ИЛИ (|), исключающее ИЛИ (^) и НЕ (~) выполняют те же функции, что и их логические аналоги, которые были рассмотрены в главе 2. Однако, в отличие от логических операторов, побитовые операторы оперируют с отдельными двоичными разрядами. Ниже приведены результаты выполнения побитовых операторов, операндами которых являются единицы и нули.

p	q	p & q	p | q	p ^ q	~p
0	0	0	0		1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

С точки зрения наиболее распространенного применения побитовую операцию И можно рассматривать как способ сброса единиц в отдельных двоичных разрядах. Это означает, что если какой-либо бит в любом из операндов равен 0, то соответствующий бит результата всегда будет нулевым.

```
  1101 0011
& 1010 1010
  ---------
  1000 0010
```

Ниже приведен пример программы, демонстрирующий применение оператора &. В этом примере строчные буквы английского алфавита преобразуются в прописные путем сброса шестого бита в коде символа. Коды строчных букв английского алфавита в кодировках ASCII и Unicode отличаются от кодов соответствующих прописных букв на величину 32. Поэтому для преобразования строчных букв в прописные достаточно сбросить в нуль шестой бит в кодах их символов.

```
// Преобразование строчных букв английского алфавита в прописные
class UpCase {
   public static void main(String args[]) {
      char ch;

      for(int i=0; i < 10; i++) {
         ch = (char) ('a' + i);
         System.out.print(ch);

         // В следующем операторе сбрасывается шестой бит.
         // После этого в переменной ch будет храниться код
         // символа прописной буквы.
         ch = (char) ((int) ch & 65503);

         System.out.print(ch + " ");
      }
   }
}
```

Результат выполнения данной программы выглядит следующим образом:

```
aA bB cC dD eE fF gG hH iI jJ
```

Значение `65503`, используемое в побитовой операции И, является десятичным представлением двоичного числа `1111111111011111`. Таким образом, при выполнении данной операции все биты кода символа в переменной `ch`, за исключением шестого, остаются прежними, тогда как шестой бит сбрасывается в нуль.

Побитовая операция И удобна и в том случае, когда требуется выяснить, установлен или сброшен отдельный бит числа. Например, в приведенной ниже строке кода проверяется, установлен ли четвертый бит значения переменной `status`:

```
if(status & 8) System.out.println("бит 4 установлен");
```

Выбор числа 8 обусловлен тем, что в данном примере нас интересует состояние четвертого бита в переменной `status`, а в двоичном представлении числа 8 все биты, кроме четвертого, нулевые. Таким образом, в условной инструкции `if` логическое значение `true` будет получено только в том случае, если четвертый бит значения переменной `status` также установлен в единицу. Подобный подход можно применить и для преобразования значения типа `byte` в двоичный формат.

```
// Отображение битов, составляющих байт
class ShowBits {
   public static void main(String args[]) {
      int t;
      byte val;

      val = 123;
      for(t=128; t > 0; t = t/2) {
```

```
            if((val & t) != 0) System.out.print("1 ");
            else System.out.print("0 ");
        }
    }
}
```

Выполнение этой программы дает следующий результат:

```
0 1 1 1 1 0 1 1
```

Здесь в цикле `for` последовательно проверяется каждый бит значения переменной `val`. Для выяснения того, установлен ли бит, выполняется побитовая операция И. Если бит установлен, отображается цифра `1`, иначе — `0`. В упражнении 5.3 будет показано, как расширить этот элементарный пример для создания класса, в котором будут отображаться биты двоичного представления целого числа любого типа.

Побитовая операция ИЛИ выполняет действия, противоположные побитовой операции И, и служит для установки отдельных битов. Любой бит, значение которого равно единице хотя бы в одном из двух операндов, в результирующем значении будет установлен в 1.

```
  1101 0011
| 1010 1010
  ---------
  1111 1011
```

Побитовую операцию ИЛИ можно использовать для преобразования прописных букв английского алфавита в строчные. Ниже приведен пример программы, решающей эту задачу.

```
// Преобразование прописных букв английского алфавита в строчные
class LowCase {
    public static void main(String args[]) {
        char ch;

        for(int i=0; i < 10; i++) {
            ch = (char) ('A' + i);
            System.out.print(ch);

            // В результате установки в единицу шестого бита
            // значения переменной ch она всегда будет
            // содержать прописную букву
            ch = (char) ((int) ch | 32);

            System.out.print(ch + " ");
        }
    }
}
```

В результате выполнения этой программы будет получен следующий результат:

```
Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj
```

В приведенном выше примере операндами побитовой операции **ИЛИ** являются код символа и значение 32 (двоичное представление — 0000000000100000). Как видите, в двоичном представлении значения 32 установлен только шестой бит. Используя это значение в качестве одного операнда в побитовой операции **ИЛИ** с любым другим значением в качестве другого операнда, получим результат, в котором устанавливается шестой бит, а состояние всех остальных битов остается без изменения. Таким образом, любая прописная буква будет преобразована в строчную.

Побитовая операция исключающего **ИЛИ** дает результат, в котором отдельный бит устанавливается (становится равным 1) в том и только в том случае, если соответствующие биты в двух операндах имеют разные значения. Ниже приведен пример выполнения побитовой операции исключающего **ИЛИ**.

```
    0111 1111
^   1011 1001
   ----------
    1100 0110
```

Побитовая операция исключающего **ИЛИ** имеет одну интересную особенность, которая позволяет очень легко шифровать сообщения. Если выполнить данную операцию сначала над некоторыми значениями `X` и `Y`, а затем над ее результатом и значением `Y`, то мы снова получим значение `X`. Например, при выполнении приведенной ниже последовательности операций переменная `R2` получит то же значение, что и `X`. Таким образом, последовательное применение двух побитовых операций исключающего **ИЛИ** восстанавливает исходное значение:

```
R1 = X ^ Y; R2 = R1 ^ Y;
```

Эту особенность побитовой операции исключающего **ИЛИ** можно использовать для создания простейшей шифрующей программы, в которой некоторое целое число будет использоваться в качестве ключа, применяемого как в процессе шифрования, так и дешифрования сообщений. Над всеми символами сообщения и данным числом будет выполняться побитовая операция исключающего **ИЛИ**. Сначала данная операция будет выполняться при шифровании, формируя зашифрованный текст, а затем при дешифровании, которое восстановит исходный текст сообщения. Ниже приведен пример простой программы, выполняющей шифрование и дешифрование коротких сообщений.

```
// Использование побитовой операции исключающего ИЛИ
// для шифрования и дешифрования сообщений
class Encode {
```

```
    public static void main(String args[]) {
        String msg = "Это просто тест";
        String encmsg = "";
        String decmsg = "";
        int key = 88;

        System.out.print("Исходное сообщение: ");
        System.out.println(msg);

        // Шифрование сообщения
        for(int i=0; i < msg.length(); i++)
            encmsg = encmsg + (char) (msg.charAt(i) ^ key);

        System.out.print("Зашифрованное сообщение: ");
        System.out.println(encmsg);

        // Дешифровка сообщения
        for(int i=0; i < msg.length(); i++)
            decmsg = decmsg + (char) (encmsg.charAt(i) ^ key);

        System.out.print("Дешифрованное сообщение: ");
        System.out.println(decmsg);
    }
}
```


В результате выполнения этой программы будет получен следующий результат.

```
Исходное сообщение: Это просто текст
Зашифрованное сообщение: 01+x1+x9x,=+,
Дешифрованное сообщение: Это просто текст
```

Как видите, в результате двух побитовых операций исключающего ИЛИ с одним и тем же ключом получается дешифрованное сообщение, совпадающее с исходным.

Унарная побитовая операция НЕ (или дополнение до 1) изменяет на обратное состояние всех битов операнда. Так, если некоторая целочисленная переменная A содержит значение с двоичным представлением 10010110, то в результате выполнения побитовой операции ~A получится двоичная комбинация 01101001.

Ниже приведен пример программы, демонстрирующий применение побитовой операции НЕ. Эта программа отображает число и его дополнение в двоичном представлении.

```
// Демонстрация побитовой операции НЕ
class NotDemo {
    public static void main(String args[]) {
        byte b = -34;

        for(int t=128; t > 0; t = t/2) {
```

```
            if((b & t) != 0) System.out.print("1 ");
            else System.out.print("0 ");
        }
        System.out.println();

        // Обращение состояния всех битов
        b = (byte) ~b;

        for(int t=128; t > 0; t = t/2) {
            if((b & t) != 0) System.out.print("1 ");
            else System.out.print("0 ");
        }
    }
}
```

В результате выполнения этой программы будет получен следующий результат:

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

Операции побитового сдвига

В Java предусмотрена возможность сдвига битов, составляющих числовое значение, влево или вправо на заданное количество позиций. Для этой цели в Java имеются три перечисленных ниже оператора сдвига.

<<	Сдвиг влево
>>	Сдвиг вправо
>>>	Сдвиг вправо без знака

Ниже приведен общий синтаксис этих операторов.

```
значение << число_битов
значение >> число_битов
значение >>> число_битов
```

Здесь *число_битов* — это число позиций двоичных разрядов, на которое сдвигается указанное значение.

При сдвиге влево освободившиеся младшие разряды заполняются нулями, а при сдвиге вправо дело обстоит немного сложнее. Как известно, признаком отрицательного целого числа является единица в старшем разряде, поэтому при сдвиге вправо старший (знаковый) разряд сохраняется. Если число положительное, то в него записывается нуль, а если отрицательное — единица.

Помимо сохранения знакового разряда, необходимо помнить еще об одной особенности операции сдвига вправо. Отрицательные числа в Java (как, впрочем, и в других языках программирования) представляются в виде *дополнения до двух*. Для того чтобы преобразовать положительное число в отрицательное,

нужно изменить на обратное состояние всех битов его двоичного представления и к полученному результату прибавить единицу. Так, значение –1 имеет байтовое представление 11111111. Сдвинув это значение вправо на любое число позиций, мы снова получим –1!

Если при сдвиге вправо не требуется сохранять знаковый разряд, то можно воспользоваться операцией сдвига вправо без знака (>>>). В этом случае освободившиеся старшие разряды всегда будут заполняться нулями. Именно поэтому такую операцию иногда называют сдвигом *с заполнением нулями*. Сдвигом вправо без знака удобно пользоваться для обработки нечисловых значений, в том числе кодов состояния.

При выполнении любого сдвига те биты, которые выходят за пределы ячейки памяти, теряются. Циклический сдвиг в Java не поддерживается, и поэтому восстановить утерянные разряды невозможно.

Ниже приведен пример программы, демонстрирующий эффект применения операций сдвига влево и вправо. В двоичном представлении исходного целочисленного значения 1 установлен лишь младший разряд. К этому значению восемь раз применяется операция сдвига влево. После каждого сдвига на экран выводится восемь младших разрядов числа. Затем единица устанавливается в восьмом двоичном разряде числа, и выполняются его сдвиги вправо.

```
// Демонстрация использования операторов << и >>
class ShiftDemo {
   public static void main(String args[]) {
      int val = 1;

      for(int i = 0; i < 8; i++) {
         for(int t=128; t > 0; t = t/2) {
            if((val & t) != 0) System.out.print("1 ");
            else System.out.print("0 ");
         }
         System.out.println();
         val = val << 1; // сдвиг влево
      }
      System.out.println();

      val = 128;
      for(int i = 0; i < 8; i++) {
         for(int t=128; t > 0; t = t/2) {
            if((val & t) != 0) System.out.print("1 ");
            else System.out.print("0 ");
         }
         System.out.println();
         val = val >> 1; // сдвиг вправо
      }
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Выполняя сдвиг значений типа `byte` и `short`, необходимо соблюдать осторожность, поскольку исполняющая среда Java автоматически преобразует их в тип `int` и лишь потом вычисляет выражение с операцией сдвига. Так, если сдвинуть вправо значение типа `byte`, оно будет сначала повышено до типа `int`, а результат сдвига будет также отнесен к типу `int`. Обычно такое преобразование не влечет за собой никаких последствий. Но если попытаться сдвинуть отрицательное значение типа `byte` или `short`, то при повышении до типа `int` оно будет дополнено знаком, а следовательно, старшие его разряды будут заполнены единицами. Это вполне оправдано при обычном сдвиге вправо. Но при выполнении сдвига с заполнением нулями в байтовом представлении числа неожиданно появятся 24 единицы, которые придется дополнительно сдвинуть, прежде чем в нем появятся нули.

Спросим у эксперта

ВОПРОС. Известно, что последовательные разряды двоичного представления числа соответствуют возрастающим степеням двойки. Значит ли это, что операторы сдвига можно использовать для умножения или деления числа на два?

ОТВЕТ. Совершенно верно. Операторы сдвига часто используются именно для этой цели. При сдвиге влево число умножается на два, а при сдвиге вправо — делится на два. Нужно лишь не забывать о том, что при сдвиге могут исчезнуть установленные биты, что приведет к потере точности.

Побитовые составные операторы присваивания

Для всех двоичных побитовых операций имеются соответствующие составные операторы присваивания. Например, в двух приведенных ниже строках кода переменной x присваивается результат выполнения операции исключающего ИЛИ, операндами которой служат первоначальное значение переменной x и числовое значение 127.

```
x = x ^ 127;
x ^= 127;
```

Упражнение 5.3

Создание класса ShowBits

ShowBitsDemo.java

В данном упражнении вам предстоит создать класс ShowBits, который позволит отображать произвольное целочисленное значение в двоичном виде. Этот класс может вам очень пригодиться при разработке некоторых программ. Так, если требуется отладить код драйвера устройства, возможность контролировать поток данных в двоичном виде будет весьма кстати. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл ShowBitsDemo.java.
2. Создайте класс ShowBits, начав его со следующего кода.

```
class ShowBits {
   int numbits;

   ShowBits(int n) {
      numbits = n;
   }
```

Конструктор класса ShowBits позволяет создавать объекты, отображающие заданное число битов. Например, для создания объекта, отображающего 8 младших битов некоторого значения, служит следующее выражение:

```
ShowBits byteval = new ShowBits(8)
```

Число битов, которые должны отображаться, сохраняется в переменной экземпляра numbits.

3. Для вывода двоичных значений в классе ShowBits определен метод show(), код которого приведен ниже.

```
void show(long val) {
   long mask = 1;

   // Сдвиг значения 1 влево на нужную позицию
   mask <<= numbits-1;

   int spacer = 0;
   for(; mask != 0; mask >>>= 1) {
```

```
        if((val & mask) != 0) System.out.print("1");
        else System.out.print("0");
        spacer++;
        if((spacer % 8) == 0) {
           System.out.print(" ");
           spacer = 0;
        }
     }
     System.out.println();
  }
```

Обратите внимание на то, что данному методу передается один параметр типа `long`. Но это вовсе не означает, что при вызове ему нужно всегда передавать значение типа `long`. Правила автоматического преобразования типов в Java допускают передавать методу `show()` любое целочисленное значение, а количество отображаемых битов определяется переменной `numbits`. Группы из 8 битов разделяются в методе `show()` пробелами. Это упрощает чтение длинных двоичных комбинаций.

4. Ниже приведен полный исходный код программы, содержащейся в файле `ShowBitsDemo.java`.

```
/*
   Упражнение 5.3

   Создание класса для отображения значений в двоичном виде
*/

class ShowBits {
   int numbits;

   ShowBits(int n) {
      numbits = n;
   }

   void show(long val) {
      long mask = 1;

      // Сдвиг значения 1 влево на нужную позицию
      mask <<= numbits-1;

      int spacer = 0;
      for(; mask != 0; mask >>>= 1) {
         if((val & mask) != 0) System.out.print("1");
         else System.out.print("0");
         spacer++;
         if((spacer % 8) == 0) {
            System.out.print(" ");
            spacer = 0;
         }
      }
```

```
      System.out.println();
    }
  }

  // Демонстрация использования класса ShowBits
  class ShowBitsDemo {
    public static void main(String args[]) {
      ShowBits b = new ShowBits(8);
      ShowBits i = new ShowBits(32);
      ShowBits li = new ShowBits(64);

      System.out.println("123 в двоичном представлении: ");
      b.show(123);

      System.out.println("\n87987 в двоичном представлении: ");
      i.show(87987);

      System.out.println("\n237658768 в двоичном представлении: ");
      li.show(237658768);

      // Можно также отобразить младшие
      // разряды любого целого числа
      System.out.println("\nМладшие 8 битов числа 87987
                          в двоичном представлении: ");
      b.show(87987);
    }
  }
```

5. **Результат выполнения программы** ShowBitsDemo **выглядит следующим образом.**

```
123 в двоичном представлении:
01111011

87987 в двоичном представлении:
00000000 00000001 01010111 10110011

237658768 в двоичном представлении:
00000000 00000000 00000000 00000000 00001110 00101010 01100010
10010000

Младшие 8 битов числа 87987 в двоичном представлении:
10110011
```

Оператор ?

Оператор ? — один из самых удобных в Java и часто используется вместо инструкций if-else следующего вида:

```
if (условие)
  переменная = выражение_1;
```

```
else
   переменная = выражение_2;
```

Здесь значение, присваиваемое *переменной*, определяется условием инструкции `if`.

Оператор `?` называется ***тернарным***, поскольку он обрабатывает три операнда. Этот оператор записывается в следующей общей форме:

```
выражение_1 ? выражение_2 : выражение_3;
```

Здесь *выражение_1* должно быть логическим, т.е. возвращать тип `boolean`, а *выражение_2* и *выражение_3*, разделяемые двоеточием, могут быть любого типа, за исключением `void`. Но типы второго и третьего выражений должны совпадать.

Значение выражения `?` определяется следующим образом. Сначала вычисляется *выражение_1*. Если оно дает логическое значение `true`, то вычисляется *выражение_2*, а его значение становится результирующим для всего оператора `?`. Если же *выражение_1* дает логическое значение `false`, то вычисляется *выражение_3*, а его значение становится результирующим для всего оператора `?`. Рассмотрим пример, в котором сначала вычисляется абсолютное значение переменной `val`, а затем оно присваивается переменной `absval`:

```
absval = val < 0 ? -val : val; // получить абсолютное значение
                               // переменной val
```

В данном примере переменной `absval` присваивается значение переменной `val`, если это значение больше или равно 0. А если значение переменной `val` отрицательное, то переменной `absval` присваивается значение `val` со знаком "минус", что в итоге дает положительную величину. Код, выполняющий ту же самую задачу, но с помощью логической конструкции `if-else`, будет выглядеть следующим образом.

```
if(val < 0) absval = -val;
else absval = val;
```

Рассмотрим еще один пример применения оператора `?`. В этом примере программы выполняется деление двух чисел, но не допускается деление на нуль.

```
// Предотвращение деления на нуль с помощью оператора ?
class NoZeroDiv {
   public static void main(String args[]) {
      int result;

      for(int i = -5; i < 6; i++) {
         result = i != 0 ? 100 / i : 0; ◄——— Деление на нуль предотвращается
         if(i != 0)
            System.out.println("100 / " + i + " равно " + result);
      }
   }
}
```

Ниже приведен результат выполнения данной программы.

```
100 / -5 равно -20
100 / -4 равно -25
100 / -3 равно -33
100 / -2 равно -50
100 / -1 равно -100
100 / 1 равно 100
100 / 2 равно 50
100 / 3 равно 33
100 / 4 равно 25
100 / 5 равно 20
```

Обратите внимание на следующую строку кода:

```
result = i != 0 ? 100 / i : 0;
```

где переменной `result` присваивается результат деления числа 100 на значение переменной `i`. Но деление выполняется только в том случае, если значение переменной `i` не равно нулю. В противном случае переменной `result` присваивается нулевое значение.

Значение, возвращаемое оператором `?`, не обязательно присваивать переменной. Его можно, например, использовать в качестве параметра при вызове метода. Если же все три выражения оператора `?` имеют тип `boolean`, то сам оператор `?` может быть использован в качестве условия для выполнения цикла или инструкции `if`. Ниже приведена немного видоизмененная версия предыдущего примера программы. Ее выполнение дает такой же результат, как и прежде.

```
// Предотвращение деления на нуль с помощью оператора ?
class NoZeroDiv2 {
   public static void main(String args[]) {

      for(int i = -5; i < 6; i++)
        if(i != 0 ? true : false)
          System.out.println("100 / " + i + " равно " + 100 / i);
   }
}
```

Обратите внимание на выражение, определяющее условие выполнения инструкции `if`. Если значение переменной `i` равно нулю, то оператор `?` возвращает логическое значение `false`, что предотвращает деление на нуль, и результат не отображается. В противном случае осуществляется обычное деление.

Вопросы и упражнения для самопроверки

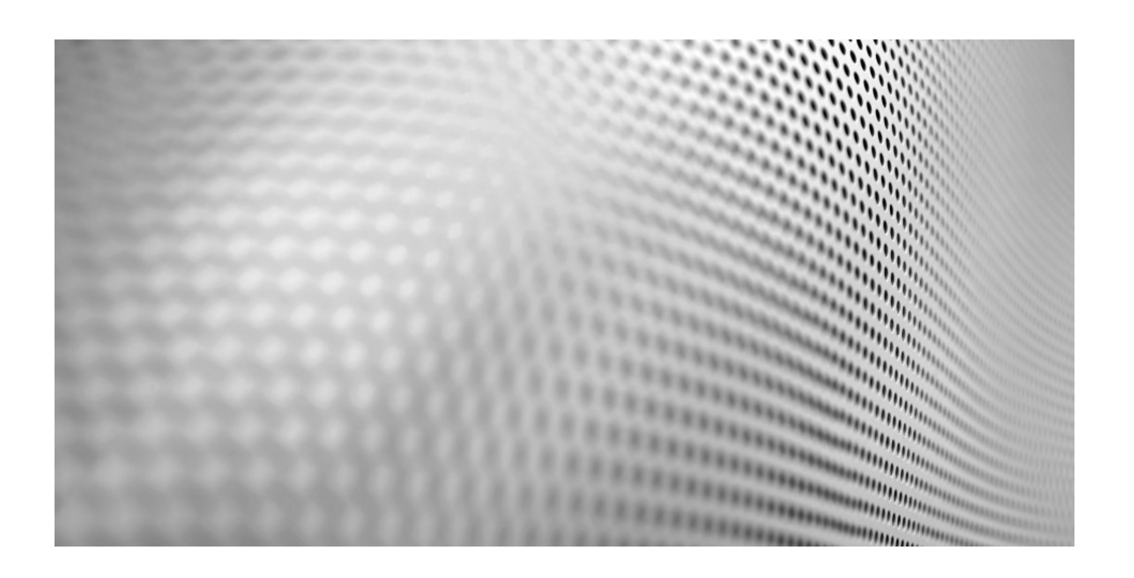
1. Продемонстрируйте два способа объявления одномерного массива, состоящего из 12 элементов типа `double`.
2. Покажите, как инициализировать одномерный массив целочисленными значениями от 1 до 5.
3. Напишите программу, в которой массив используется для нахождения среднего арифметического десяти значений типа `double`. Используйте любые десять чисел.
4. Измените программу, написанную в упражнении 5.1, таким образом, чтобы она сортировала массив строк. Продемонстрируйте ее работоспособность.
5. В чем отличие методов `indexOf()` и `lastIndexOf()` класса `String`?
6. Все символьные строки являются объектами типа `String`. Покажите, как вызываются методы `length()` и `charAt()` для строкового литерала `"Мне нравится Java"`.
7. Расширьте класс `Encode` таким образом, чтобы в качестве ключа шифрования использовалась строка из восьми символов.
8. Можно ли применять побитовые операции к значениям типа `double`?
9. Перепишите приведенную ниже последовательность инструкций, воспользовавшись оператором `?`.

```
if(x < 0) y = 10;
else y = 20;
```

10. В приведенном ниже фрагменте кода содержится знак `&`. Какой операции он соответствует: побитовой или логической? Обоснуйте свой ответ.

```
boolean a, b;
// ...
if(a & b) ...
```

11. Является ли ошибкой превышение верхней границы массива? Является ли ошибкой использование отрицательных значений для доступа к элементам массива?
12. Как обозначается операция сдвига вправо без знака?
13. Перепишите рассмотренный ранее класс `MinMax` таким образом, чтобы в нем использовался цикл типа `for-each`.
14. В упражнении 5.1 была реализована пузырьковая сортировка. Можно ли в программе из этого примера заменить обычный цикл `for` циклом типа `for-each`? Если нельзя, то почему?
15. Можно ли управлять инструкцией `switch` с помощью объектов типа `String`?



Глава 6

Подробнее о методах и классах

В этой главе...

- Управление доступом к членам классов
- Передача объектов при вызове методов
- Возврат объектов из методов
- Перегрузка методов
- Перегрузка конструкторов
- Рекурсия
- Использование ключевого слова `static`
- Применение внутренних классов
- Использование переменного числа аргументов

В этой главе мы перейдем к углубленному рассмотрению классов и методов. Сначала будет показано, каким образом контролируется доступ к членам класса, а затем будут рассмотрены особенности передачи и возврата объектов из методов, детали перегрузки методов, использования рекурсии и ключевого слова `static`. Кроме того, будут представлены вложенные классы и методы с переменным числом аргументов.

Управление доступом к членам класса

Поддержка инкапсуляции в классе обеспечивает два основных преимущества. Во-первых, класс связывает данные с кодом. Это использовалось в предыдущих примерах программ, начиная с главы 4. И во-вторых, класс предоставляет средства для управления доступом к его членам. Именно эта, вторая, особенность и будет рассмотрена в данной главе.

В Java имеются два типа членов класса: открытые (public) и закрытые (private), хотя в действительности дело обстоит немного сложнее. Доступ к открытому члену свободно осуществляется из кода, определенного вне класса. Именно этот тип членов класса использовался в рассмотренных до сих пор примерах программ. Закрытый член класса доступен только методам, определенным в самом классе. С помощью закрытых членов и организуется управление доступом.

Ограничение доступа к членам класса является основополагающей концепцией объектно-ориентированного программирования, поскольку это позволяет исключить неверное использование объекта. Разрешая доступ к закрытым данным только с помощью строго определенного ряда методов, можно

предупредить присваивание неверных значений этим данным, выполняя, например, проверку диапазона представления чисел. Для закрытого члена класса нельзя задать значение непосредственно в коде за пределами класса. Но в то же время можно полностью управлять тем, как и когда данные используются в объекте. Следовательно, корректно реализованный класс образует некий “черный ящик”, которым можно пользоваться, но внутренний механизм его действия закрыт для вмешательства извне.

В рассмотренных ранее примерах программ не уделялось особого внимания управлению доступом, поскольку в Java члены класса по умолчанию доступны из остальных частей программы. (Иными словами, они открыты для доступа по умолчанию.) Это удобно для создания небольших программ (в том числе и тех, что служат примерами в данной книге), но обычно недопустимо в реальных условиях эксплуатации программного обеспечения. Ниже будет показано, какими языковыми средствами Java можно пользоваться для управления доступом.

Модификаторы доступа в Java

Управление доступом к членам класса в Java осуществляется с помощью трех *модификаторов доступа*: `public`, `private` и `protected`. Если модификатор не указан, то используется тип доступа по умолчанию. В этой главе будут рассмотрены модификаторы `public` и `private`. Модификатор `protected` непосредственно связан с наследованием и поэтому будет обсуждаться в главе 8.

Когда член класса помечается модификатором `public`, он становится доступным из любого другого кода в программе, включая и методы, определенные в других классах. Если же член класса обозначается модификатором `private`, то он может быть доступен только другим членам этого класса. Следовательно, методы из других классов не имеют доступа к закрытому члену класса.

Если все классы в программе относятся к одному пакету, то отсутствие модификатора доступа равнозначно указанию модификатора `public` по умолчанию. *Пакет* представляет собой группу классов, предназначенных как для структурирования классов, так и для управления доступом. Рассмотрение пакетов мы отложим до главы 8, а для примеров программ, представленных в этой и предыдущих главах, тип доступа по умолчанию — `public`.

Модификатор доступа указывается перед спецификацией отдельного члена класса. Это означает, что именно с него должна начинаться инструкция объявления члена класса. Вот несколько примеров.

```
public String errMsg;
private accountBalance bal;

private boolean isError(byte status) { // ...
```

Для того чтобы стал понятнее эффект от применения модификаторов доступа `public` и `private`, рассмотрим следующий пример программы.

```
// Сравнение модификаторов доступа public и private
class MyClass {
   private int alpha;  // закрытый доступ
   public int beta;    // открытый доступ
   int gamma; // тип доступа по умолчанию (по сути, public)

   // Методы доступа к переменной alpha. Члены класса могут
   // обращаться к закрытым членам того же класса.
   void setAlpha(int a) {
      alpha = a;
   }

   int getAlpha() {
      return alpha;
   }
}

class AccessDemo {
   public static void main(String args[]) {
      MyClass ob = new MyClass();

      // Доступ к переменной alpha возможен только с помощью
      // специально предназначенных для этой цели методов
      ob.setAlpha(-99);
      System.out.println("ob.alpha: " + ob.getAlpha());

      // Обращение к переменной alpha так, как показано ниже,
      // недопустимо
      // ob.alpha = 10; // Ошибка: alpha - закрытая переменная! ◄— Ошибка, поскольку alpha — закрытая переменная!

      // Следующие обращения вполне допустимы, так как
      // переменные beta и gamma являются открытыми
      ob.beta = 88;  ◄——— Допустимо, поскольку это открытые переменные
      ob.gamma = 99;
   }
}
```

Как видите, в классе MyClass переменная alpha определена как private, переменная beta — как public, а перед переменной gamma модификатор доступа отсутствует, т.е. в данном примере она ведет себя как открытый член класса, которому по умолчанию присваивается модификатор доступа public. Переменная alpha закрыта, и поэтому к ней невозможно обратиться за пределами ее класса. Следовательно, в классе AccessDemo нельзя пользоваться переменной alpha непосредственно. Доступ к ней организуется с помощью открытых методов setAlpha() и getAlpha(), определенных в одном с ней классе. Если удалить комментарии в начале следующей строки кода, то скомпилировать программу не удастся:

```
// ob.alpha = 10; // Ошибка: alpha - закрытая переменная!
```

Компилятор выдаст сообщение об ошибке, связанной с нарушением правил доступа. Несмотря на то что переменная alpha недоступна для кода за пределами класса MyClass, пользоваться ею можно с помощью открытых методов доступа setAlpha() и getAlpha().

Таким образом, закрытые переменные могут быть свободно использованы другими членами класса, но недоступны за пределами этого класса.

Рассмотрим практическое применение средств управления доступом на примере приведенной ниже программы. Во время ее выполнения предотвращается возникновение ошибок выхода за пределы массива. Это достигается следующим образом. Массив объявляется как закрытый член класса, а доступ к нему осуществляется с помощью методов, специально предназначенных для этой цели. Эти методы отслеживают попытки обращения к элементам, не входящим в массив, и вместо аварийного завершения программы возвращают сообщение об ошибке. Массив определяется в классе FailSoftArray, код которого приведен ниже.

```
/* В этом классе реализуется "отказоустойчивый" массив,
   предотвращающий ошибки времени выполнения
*/
class FailSoftArray {
   private int a[];     // ссылка на массив
   private int errval;  // значение, возвращаемое в случае
                        // возникновения ошибки при выполнении
                        // метода get()
   public int length;   // открытая переменная length

   // Конструктору данного класса передаются размер массива
   // и значение, которое должен возвращать метод get() при
   // возникновении ошибки
   public FailSoftArray(int size, int errv) {
      a = new int[size];
      errval = errv;
      length = size;
   }

   // Возврат значения элемента массива с заданным индексом
   public int get(int index) {
      if(indexOK(index)) return a[index]; ◄── Отслеживание попытки
      return errval;                           выхода за пределы массива
   }

   // Установка значения элемента с заданным индексом.
   // Если возникнет ошибка, вернуть логическое значение false.
   public boolean put(int index, int val) {
      if(indexOK(index)) { ◄──
         a[index] = val;
         return true;
      }
      return false;
   }
```

```
   // Возврат логического значения true, если индекс
   // не выходит за пределы массива
   private boolean indexOK(int index) {
      if(index >= 0 & index < length) return true;
      return false;
   }
}

// Демонстрация работы с "отказоустойчивым" массивом
class FSDemo {
   public static void main(String args[]) {
      FailSoftArray fs = new FailSoftArray(5, -1);
      int x;

      // Демонстрация корректной обработки ошибок
      System.out.println("Обработка ошибок без вывода отчета.");
      for(int i=0; i < (fs.length * 2); i++)
         fs.put(i, i*10); ◄── Для обращения к элементам массива должны использоваться
                               его методы доступа
      for(int i=0; i < (fs.length * 2); i++) {
         x = fs.get(i); ◄──
         if(x != -1) System.out.print(x + " ");
      }
      System.out.println("");

      // Обработка ошибок
      System.out.println("\nОбработка ошибок с выводом отчета.");
      for(int i=0; i < (fs.length * 2); i++)
         if(!fs.put(i, i*10))
         System.out.println("Индекс " + i +
                            " вне допустимого диапазона");

      for(int i=0; i < (fs.length * 2); i++) {
         x = fs.get(i);
         if(x != -1) System.out.print(x + " ");
         else
            System.out.println("Индекс " + i +
                               " вне допустимого диапазона");
      }
   }
}
```

В результате выполнения этой программы будут выведены следующие строки.

```
Обработка ошибок без вывода отчета.
0 10 20 30 40

Обработка ошибок с выводом отчета.
Индекс 5 вне допустимого диапазона
Индекс 6 вне допустимого диапазона
Индекс 7 вне допустимого диапазона
Индекс 8 вне допустимого диапазона
Индекс 9 вне допустимого диапазона
0 10 20 30 40 Индекс 5 вне допустимого диапазона
Индекс 6 вне допустимого диапазона
```

```
Индекс 7 вне допустимого диапазона
Индекс 8 вне допустимого диапазона
Индекс 9 вне допустимого диапазона
```

А теперь рассмотрим этот пример подробнее. В классе `FailSoftArray` определены три закрытых члена. Первым из них является переменная `a`, в которой содержится ссылка на массив, предназначенный для хранения данных. Вторым членом является переменная `errval`, в которой хранится значение, возвращаемое вызывающей части программы в том случае, если вызов метода `get()` приводит к ошибке. И третьим членом является метод `indexOk()`, в котором определяется, находится ли индекс в допустимых пределах. Эти три члена могут быть использованы только другими членами класса `FailSoftArray`. Остальные члены данного класса объявлены открытыми и могут быть вызваны из любой части программы, в которой используется класс `FailSoftArray`.

При создании объекта типа `FailSoftArray` следует указать размер массива и значение, которое должно возвращаться в случае неудачного вызова метода `get()`. Это значение не может совпадать ни с одним значением, хранящимся в массиве. После создания объекта непосредственный доступ извне его к массиву, на который указывает ссылка, хранящаяся в переменной `a`, а также к переменной `errval` невозможен, что исключает их некорректное использование. В частности, пользователь не сможет непосредственно обратиться к массиву по ссылке в переменной `a`, задав индекс элемента, выходящий за границы допустимого диапазона. Доступ к указанным элементам возможен только с помощью методов `get()` и `put()`.

Метод `indexOK()` объявлен как закрытый главным образом для того, чтобы продемонстрировать управление доступом. Но даже если бы он был открытым, то это не создавало бы никакого риска, поскольку он не видоизменяет объект. Однако, поскольку этот метод используется только членами класса `FailSoftArray`, он объявлен закрытым.

Обратите внимание на то, что переменная экземпляра `length` открыта. Это согласуется с правилами реализации массивов в Java. Для того чтобы получить данные о длине массива типа `FailSoftArray`, достаточно прочитать значение переменной экземпляра `length`.

Для сохранения данных в массиве типа `FailSoftArray` по указанному индексу вызывается метод `put()`, тогда как метод `get()` извлекает содержимое элемента этого массива по заданному индексу. Если индекс оказывается вне границ массива, то метод `put()` возвращает логическое значение `false`, а метод `get()` — значение `errval`.

Ради простоты в большинстве примеров программ, представленных в этой книге, на члены класса будет в основном распространяться тип доступа по умолчанию. Но не следует забывать, что в реальных объектно-ориентированных программах очень важно ограничивать доступ к членам класса и в особенности к переменным. Как будет показано в главе 7, при наследовании роль средств управления доступом еще более возрастает.

Примечание

Влияние на доступ к членам класса также оказывают модули, которые появились в JDK 9. Они будут рассмотрены в главе 15.

Упражнение 6.1 Усовершенствование класса Queue

Queue.java

Модификатор доступа `private` можно использовать для усовершенствования класса `Queue`, разработанного в упражнении 5.2 (см. главу 5). В текущей версии этого класса используется тип доступа по умолчанию, который делает все члены данного класса открытыми. Это означает, что другие классы могут непосредственно обращаться к элементам базового массива — и даже вне очереди. А поскольку назначение класса, реализующего очередь, состоит в том, чтобы обеспечить принцип доступа "первым пришел — первым обслужен", то возможность произвольного обращения к элементам массива явно неуместна. В частности, это давало бы возможность недобросовестным программистам изменять индексы в переменных `putloc` и `getloc`, искажая тем самым организацию очереди. Подобные недостатки нетрудно устранить с помощью модификатора доступа `private`. Поэтапно процесс создания программы описан ниже.

1. Создайте новый файл `Queue.java`.
2. Снабдите массив `q`, а также переменные `putloc` и `getloc` в классе `Queue` модификаторами доступа `private`. В результате код этого класса должен выглядеть так.

```
// Усовершенствованный класс очереди, предназначенной
// для хранения символьных значений
class Queue {
   // Эти члены класса теперь являются закрытыми
   private char q[]; // массив для хранения элементов очереди
   private int putloc, getloc;  // индексы для вставки и
                                // извлечения элементов очереди

   Queue(int size) {
      q = new char[size];     // выделение памяти для очереди
      putloc = getloc = 0;
   }

   // Помещение символа в очередь
   void put(char ch) {
      if(putloc==q.length-1) {
         System.out.println(" - Очередь заполнена.");
         return;
      }

      q[putloc++] = ch;
   }
```

```
    // Извлечение символа из очереди
    char get() {
       if(getloc == putloc) {
          System.out.println(" - Очередь пуста.");
          return (char) 0;
       }

       return q[getloc++];
    }
}
```

3. Изменение типа доступа к массиву `q` и переменным `putloc` и `getloc` с выбираемого по умолчанию на закрытый (`private`) никак не скажется на работе программ, в которых класс `Queue` используется корректно. В частности, этот класс будет по-прежнему нормально взаимодействовать с классом `QDemo` из упражнения 5.2. В то же время некорректное обращение к классу `Queue` станет невозможным. Например, следующий фрагмент кода не будет скомпилирован.

```
Queue test = new Queue(10);

test.q[0] = 99;       // Ошибка!
test.putloc = -100;  // Не пройдет!
```

4. Теперь, когда массив `q` и переменные `putloc` и `getloc` объявлены как `private`, класс `Queue` строго следует принципу "первым пришел — первым обслужен", в соответствии с которым действует очередь.

Передача объектов методам

Вплоть до этого момента в примерах программ при передаче параметров методам использовались лишь простые типы. Но параметрами могут быть и объекты. Например, в приведенной ниже программе определен класс `Block`, предназначенный для хранения размеров параллелепипеда в трех измерениях.

```
// Методам можно передавать объекты
class Block {
   int a, b, c;
   int volume;

   Block(int i, int j, int k) {
      a = i;
      b = j;
      c = k;
      volume = a * b * c;
   }

   // Возврат логического значения true, если
   // параметр ob определяет тот же параллелепипед
```

```
   boolean sameBlock(Block ob) {  ◄──── Использование объектного типа в качестве параметра
      if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
      else return false;
   }

   // Возврат логического значения true, если
   // параметр ob определяет параллелепипед того же объема
   boolean sameVolume(Block ob) {  ◄
      if(ob.volume == volume) return true;
      else return false;
   }
}

class PassOb {
   public static void main(String args[]) {
      Block ob1 = new Block(10, 2, 5);
      Block ob2 = new Block(10, 2, 5);
      Block ob3 = new Block(4, 5, 5);

      System.out.println("ob1 имеет те же размеры, что и ob2: " +
                         ob1.sameBlock(ob2));  ◄──── Передача объекта
      System.out.println("ob1 имеет те же размеры, что и ob3: " +
                         ob1.sameBlock(ob3));  ◄
      System.out.println("ob1 имеет тот же объем, что и ob3: " +
                         ob1.sameVolume(ob3));  ◄
   }
}
```

Выполнение этой программы приводит к следующему результату.

```
ob1 имеет те же размеры, что и ob2: true
ob1 имеет те же размеры, что и ob3: false
ob1 имеет тот же объем, что и ob3: true
```

В методах `sameBlock()` и `sameVolume()` объект `Block`, переданный им в качестве параметра, сравнивается с текущим объектом. Метод `sameBlock()` возвращает логическое значение `true` только в том случае, если все три размера обоих параллелепипедов совпадают. В методе же `sameVolume()` сравниваются лишь объемы двух параллелепипедов. Но в обоих случаях параметр `ob` имеет тип `Block`. Несмотря на то что `Block` — это класс, параметры данного типа используются точно так же, как и параметры встроенных в Java типов данных.

Способы передачи аргументов методу

Как показывает приведенный выше пример, передача объекта методу не вызывает затруднений. Однако имеются некоторые нюансы, не нашедшие отражения в данном примере. В некоторых случаях последствия передачи объекта по ссылке будут отличаться от тех результатов, к которым приводит передача значения обычного типа. Для выяснения причин этих отличий рассмотрим два возможных способа передачи аргументов методу.

Первый из них — это *вызов по значению*. В таком случае в формальный параметр метода копируется *значение* аргумента. Следовательно, изменения, вносимые в параметр метода, никоим образом не сказываются на состоянии аргумента, используемого при вызове. Вторым способом передачи аргумента является *вызов по ссылке*. В таком случае параметру метода передается не значение аргумента, а ссылка на него. В методе данная ссылка используется для доступа к конкретному аргументу, указанному при вызове. Это означает, что изменения, вносимые в параметр, *будут* оказывать влияние на аргумент, используемый при вызове метода. Как будет показано далее, несмотря на то что в Java передача аргументов осуществляется в соответствии с механизмом вызова по значению, результирующий эффект будет разным для простых и ссылочных типов.

Если методу передается простой тип, например `int` или `double`, то он передается по значению. При этом создается копия аргумента, а то, что происходит с параметром, принимающим аргумент, не распространяется за пределы метода. Рассмотрим в качестве примера следующую программу.

```
// Простые типы данных передаются методам по значению
class Test {
   // Этот метод не может изменить значения аргументов,
   // передаваемых ему при вызове
   void noChange(int i, int j) {
      i = i + j;
      j = -j;
   }
}

class CallByValue {
   public static void main(String args[]) {
      Test ob = new Test();

      int a = 15, b = 20;

      System.out.println("a и b перед вызовом: " +
                         a + " " + b);

      ob.noChange(a, b);

      System.out.println("a и b после вызова: " +
                         a + " " + b);
   }
}
```

Ниже приведен результат выполнения данной программы.

```
a и b перед вызовом: 15 20
a и b после вызова: 15 20
```

Как видите, действия, выполняемые в теле метода `noChange()`, никак не влияют на значения переменных `a` и `b` в вызывающем методе.

Если же методу передается объект, то ситуация коренным образом меняется, поскольку объекты передаются неявно, по ссылке. Вспомните, что создание переменной, для которой в качестве типа указан класс, означает создание ссылки на объект этого класса, и именно эта ссылка передается по значению в формальный параметр при передаче ее методу. Отсюда следует, что и передаваемый аргумент, и параметр метода, как содержащие одну и ту же ссылку, будут ссылаться на один и тот же объект. Таким образом, любые изменения объекта в методе *будут* вызывать соответствующие изменения в объекте, используемом в качестве аргумента. Для примера рассмотрим следующую программу.

```
// Объекты передаются методам по ссылке
class Test {
   int a, b;

   Test(int i, int j) {
      a = i;
      b = j;
   }

   // Передача объекта методу. Теперь переменные ob.a и ob.b
   // объекта, используемого при вызове, также будут изменяться.
   void change(Test ob) {
      ob.a = ob.a + ob.b;
      ob.b = -ob.b;
   }
}

class CallByRef {
   public static void main(String args[]) {
      Test ob = new Test(15, 20);

      System.out.println("ob.a и ob.b перед вызовом: " +
                         ob.a + " " + ob.b);

      ob.change(ob);

      System.out.println("ob.a и ob.b после вызова: " +
                         ob.a + " " + ob.b);
   }
}
```

Выполнение этой программы дает следующий результат.

```
ob.a и ob.b перед вызовом: 15 20
ob.a и ob.b после вызова: 35 -20
```

Как видите, в данном случае действия в методе `change()` оказывают влияние на объект, используемый в качестве аргумента этого метода.

Спросим у эксперта

ВОПРОС. Существует ли способ передачи простого типа по ссылке?

ОТВЕТ. Явным образом этого сделать нельзя. Но в Java определен ряд классов, служащих оболочкой для простых типов. Это классы `Double`, `Float`, `Byte`, `Short`, `Integer`, `Long` и `Character`. Они не только позволяют передавать простые типы по ссылке, но и содержат ряд методов для манипулирования их значениями. Например, в классах — оболочках числовых типов содержатся методы, преобразующие двоичные значения в символьную строку, а также методы, выполняющие обратное преобразование.

Возврат объектов методами

Метод может возвращать данные любого типа, включая и типы классов. Например, объект приведенного ниже класса `ErrorMsg` можно использовать для вывода сообщений об ошибке. В этом классе имеется метод `getErrorMsg()`, который возвращает объект типа `String`, описывающий конкретную ошибку. Объект типа `String` создается на основании кода ошибки, переданного методу.

```
// Возврат объекта типа String
class ErrorMsg {
   String msgs[] = {
      "Ошибка вывода",
      "Ошибка ввода",
      "Отсутствует место на диске",
      "Выход индекса за границы диапазона"
   };

   // Возврат сообщения об ошибке
   String getErrorMsg(int i) {  ◄——— Возврат объекта типа String
      if(i >=0 & i < msgs.length)
         return msgs[i];
      else
         return "Несуществующий код ошибки";
   }
}

class ErrMsg {
   public static void main(String args[]) {
      ErrorMsg err = new ErrorMsg();

      System.out.println(err.getErrorMsg(2));
      System.out.println(err.getErrorMsg(19));
   }
}
```

Выполнение этой программы даст следующий результат.

```
Отсутствует место на диске
Несуществующий код ошибки
```

Разумеется, возвращать можно и объекты создаваемых классов. Например, приведенный ниже фрагмент кода представляет собой переработанную версию предыдущей программы, в которой создаются два класса формирования ошибок: `Err` и `ErrorInfo`. В классе `Err`, помимо кода ошибки, инкапсулируется строка описания ошибки. А в классе `ErrorInfo` содержится метод `getErrorInfo()`, возвращающий объект типа `Err`.

```
// Возврат объекта, определяемого разработчиком программы
class Err {
   String msg;   // сообщение об ошибке
   int severity; // уровень серьезности ошибки

   Err(String m, int s) {
      msg = m;
      severity = s;
   }
}

class ErrorInfo {
    String msgs[] = {
      "Ошибка вывода",
      "Ошибка ввода",
      "Отсутствует место на диске",
      "Выход индекса за границы диапазона"
   };
   int howbad[] = { 3, 3, 2, 4 };

   Err getErrorInfo(int i) {   ◄——— Возврат объекта типа Err
      if(i >=0 & i < msgs.length)
         return new Err(msgs[i], howbad[i]);
      else
         return new Err("Несуществующий код ошибки", 0);
   }
}

class ErrInfo {
   public static void main(String args[]) {
      ErrorInfo err = new ErrorInfo();
      Err e;

      e = err.getErrorInfo(2);
      System.out.println(e.msg + " уровень: " + e.severity);

      e = err.getErrorInfo(19);
      System.out.println(e.msg + " уровень: " + e.severity);
   }
}
```

Результат выполнения данной версии программы выглядит следующим образом.

```
Отсутствует место на диске уровень: 2
Несуществующий код ошибки уровень: 0
```

При каждом вызове метода `getErrorInfo()` создается новый объект типа `Err`, и ссылка на него возвращается вызывающему методу. Затем этот объект используется методом `main()` для отображения степени серьезности ошибки и текстового сообщения.

Объект, возвращенный методом, существует до тех пор, пока на него имеется хотя бы одна ссылка. Если ссылки на объект отсутствуют, он уничтожается подсистемой сборки мусора. Поэтому при выполнении программы не может возникнуть ситуация, когда объект удаляется лишь потому, что метод, в котором он был создан, завершился.

Перегрузка методов

В этом разделе речь пойдет об одном из самых интересных языковых средств Java — перегрузке методов. Несколько методов одного класса могут иметь одно и то же имя, отличаясь лишь набором параметров. Подобные методы называются *перегруженными*, а сам процесс называют *перегрузкой методов*. Перегрузка методов является одним из способов реализации принципа полиморфизма в Java.

Для того чтобы перегрузить метод, достаточно объявить его новый вариант, который отличается от уже существующих, а все остальное сделает за вас компилятор. Нужно лишь соблюсти одно условие: тип и/или число параметров в каждом из перегружаемых методов должны быть разными. Одного лишь различия в типах возвращаемых значений для этой цели недостаточно. (Информации о возвращаемом типе не всегда будет хватать Java для принятия решения о том, какой именно метод должен использоваться.) Конечно, перегружаемые методы могут иметь разные возвращаемые типы, но при вызове метода выполняется лишь тот его вариант, в котором параметры соответствуют передаваемым аргументам.

Ниже приведен простой пример программы, демонстрирующий перегрузку методов.

```
// Перегрузка методов
class Overload {
   void ovlDemo() {   ◄——————————— Первая версия
      System.out.println("Без параметров");
   }

   // Перегрузка метода ovlDemo для одного параметра типа int
   void ovlDemo(int a) {   ◄——————————— Вторая версия
      System.out.println("Один параметр: " + a);
   }
```

```
   // Перегрузка метода ovlDemo для двух параметров типа int
   int ovlDemo(int a, int b) {  ◄——— Третья версия
      System.out.println("Два параметра: " + a + " " + b);
      return a + b;
   }

   // Перегрузка метода ovlDemo для двух параметров типа double
   double ovlDemo(double a, double b) {  ◄——— Четвертая версия
      System.out.println("Два параметра типа double: " +
                         a + " "+ b);
      return a + b;
   }
}

class OverloadDemo {
   public static void main(String args[]) {
      Overload ob = new Overload();
      int resI;
      double resD;

      // Поочередный вызов всех версий метода ovlDemo()
      ob.ovlDemo();
      System.out.println();

      ob.ovlDemo(2);
      System.out.println();

      resI = ob.ovlDemo(4, 6);
      System.out.println("Результат вызова ob.ovlDemo(4, 6): " +
                         resI);
      System.out.println();

      resD = ob.ovlDemo(1.1, 2.32);
      System.out.println("Результат вызова ob.ovlDemo(1.1, 2.32): " +
                         resD);
   }
}
```

Выполнение этой программы даст следующий результат.

```
Без параметров

Один параметр: 2

Два параметра: 4 6
Результат вызова ob.ovlDemo(4, 6): 10

Два параметра типа double: 1.1 2.32
Результат вызова ob.ovlDemo(1.1, 2.32): 3.42
```

Как видите, метод `ovlDemo()` перегружается четырежды. В первой его версии параметры не предусмотрены, во второй — определен один целочисленный

параметр, в третьей — два целочисленных параметра, в четвертой — два параметра типа double. Обратите внимание на то, что первые два варианта метода ovlDemo() имеют тип void, а два других возвращают значение. Как пояснялось ранее, тип возвращаемого значения не учитывается при перегрузке методов. Следовательно, попытка определить два варианта метода ovlDemo() так, как показано ниже, приводит к ошибке.

```
// Возможен лишь один вариант метода ovlDemo(int)
void ovlDemo(int a) {  ◄──────── Возвращаемое значение нельзя
   System.out.println("Один параметр: " + a);   использовать для различения
}                                               перегружаемых методов

/* Ошибка! Невозможно существование двух версий
   перегруженного метода ovlDemo(int), отличающихся
   лишь типом возвращаемого значения.
*/
int ovlDemo(int a) {  ◄────────
   System.out.println("Один параметр: " + a);
   return a * a;
}
```

Как поясняется в комментариях к приведенному выше фрагменту кода, отличия возвращаемых типов недостаточно для перегрузки методов.

Как следует из главы 2, в Java применяется автоматическое приведение типов, которое распространяется и на типы параметров перегружаемых методов. В качестве примера рассмотрим следующий фрагмент кода.

```
/* Автоматическое преобразование типов может влиять
   на выбор перегружаемого метода.
*/
class Overload2 {
   void f(int x) {
      System.out.println("Внутри f(int): " + x);
   }

   void f(double x) {
      System.out.println("Внутри f(double): " + x);
   }
}

class TypeConv {
   public static void main(String args[]) {
      Overload2 ob = new Overload2();

      int i = 10;
      double d = 10.1;

      byte b = 99;
      short s = 10;
      float f = 11.5F;
```

```
        ob.f(i); // вызов метода ob.f(int)
        ob.f(d); // вызов метода ob.f(double)

        ob.f(b); // вызов метода ob.f(int) с преобразованием типов
        ob.f(s); // вызов метода ob.f(int) с преобразованием типов
        ob.f(f); // вызов метода ob.f(double) с преобразованием типов
    }
}
```

В результате выполнения этого фрагмента кода будет получен следующий результат.

```
Внутри f(int): 10
Внутри f(double): 10.1
Внутри f(int): 99
Внутри f(int): 10
Внутри f(double): 11.5
```

В данном примере определены только два варианта метода `f()`: один имеет параметр типа `int`, а второй — параметр типа `double`. Но передать методу `f()` можно также значение типа `byte`, `short` и `float`. Значения типа `byte` и `short` исполняющая среда Java автоматически преобразует в тип `int`. В результате будет вызван вариант метода `f(int)`. А если параметр имеет значение типа `float`, то оно преобразуется в тип `double`, и далее вызывается вариант метода `f(double)`.

Важно понимать, что автоматическое преобразование типов выполняется лишь в отсутствие прямого соответствия типов параметра и аргумента. В качестве примера ниже представлена другая версия предыдущей программы, в которой добавлен вариант метода `f()` с параметром типа `byte`.

```
// Добавление версии метода f(byte)
class Overload2 {
    void f(byte x) {   <-- Эта версия имеет параметр типа byte
        System.out.println("Внутри f(byte): " + x);
    }

    void f(int x) {
        System.out.println("Внутри f(int): " + x);
    }

    void f(double x) {
        System.out.println("Внутри f(double): " + x);
    }
}

class TypeConv {
    public static void main(String args[]) {
        Overload2 ob = new Overload2();

        int i = 10;
        double d = 10.1;
```

```
        byte b = 99;
        short s = 10;
        float f = 11.5F;

        ob.f(i); // вызов метода ob.f(int)
        ob.f(d); // вызов метода ob.f(double)

        ob.f(b); // вызов метода ob.f(byte) без преобразования типов

        ob.f(s); // вызов метода ob.f(int) с преобразованием типов
        ob.f(f); // вызов метода ob.f(double) с преобразованием типов
    }
}
```

Выполнение этой версии программы дает следующий результат:

```
Внутри f(int): 10
Внутри f(double): 10.1
Внутри f(byte): 99
Внутри f(int): 10
Внутри f(double): 11.5
```

Поскольку в данной программе предусмотрена версия метода `f()`, которая имеет параметр типа `byte`, то при вызове этого метода с аргументом типа `byte` выполняется вызов `f(byte)`, и тип `byte` автоматически не преобразуется в тип `int`.

Перегрузка методов поддерживает полиморфизм, поскольку она является одним из способов реализации парадигмы "один интерфейс — множество методов". Для того чтобы стало понятнее, как и для чего это делается, необходимо принять во внимание следующее соображение: в языках программирования, не поддерживающих перегрузку методов, каждый метод должен иметь уникальное имя. Но в ряде случаев требуется выполнять одну и ту же последовательность операций над разными типами данных. В качестве примера рассмотрим функцию, вычисляющую абсолютное значение. В языках, не поддерживающих перегрузку методов, приходится создавать несколько вариантов данной функции с именами, отличающимися хотя бы одним символом. Например, в языке C функция `abs()` возвращает абсолютное значение числа типа `int`, функция `labs()` — абсолютное значение числа типа `long`, а функция `fabs()` — абсолютное значение числа с плавающей точкой. Объясняется это тем, что в языке C не поддерживается перегрузка, и поэтому каждая из функций должна обладать своим собственным именем, несмотря на то что все они выполняют одинаковые действия. Это приводит к неоправданному усложнению написания программ. Разработчику приходится не только представлять себе действия, выполняемые функциями, но и помнить все три их имени. Такая ситуация не возникает в Java, потому что все методы, вычисляющие абсолютное значение, называются одинаково. В стандартной библиотеке Java для вычисления абсолютного значения предусмотрен метод `abs()`. Его перегрузка осуществляется

в классе `Math` для обработки значений всех числовых типов. Решение о том, какой именно вариант метода `abs()` должен быть вызван, исполняющая среда Java принимает, исходя из типа аргумента.

Главная ценность перегрузки заключается в том, что она обеспечивает доступ к группе родственных методов по общему имени. Следовательно, имя `abs` обозначает *общее выполняемое действие*, а компилятор сам выбирает *конкретный* вариант метода, исходя из имеющихся обстоятельств. Благодаря полиморфизму несколько имен сводятся к одному. Несмотря на всю простоту рассматриваемого здесь примера, продемонстрированный в нем принцип полиморфизма можно расширить, чтобы выяснить, каким образом перегрузка помогает справляться с более сложными ситуациями в программировании.

Когда метод перегружается, каждая его версия может выполнять какое угодно действие. Для установления взаимосвязи перегружаемых методов не существует какого-то одного четкого правила, но с точки зрения корректного стиля программирования перегрузка методов подразумевает подобную взаимосвязь. Следовательно, не следует использовать одно и то же имя для несвязанных друг с другом методов, хотя это и возможно. Например, имя `sqr` можно было бы выбрать для методов, возвращающих квадрат и квадратный корень числа с плавающей точкой. Но ведь это принципиально разные операции. Такое применение перегрузки методов противоречит ее первоначальному назначению. На практике перегружать следует только тесно связанные операции.

Спросим у эксперта

ВОПРОС. Программисты на Java употребляют термин *сигнатура*. Что это такое?

ОТВЕТ. Применительно к Java сигнатура обозначает имя метода и список его параметров. При перегрузке методов действует следующее правило: никакие два метода из одного класса не могут иметь одинаковые сигнатуры. При этом следует иметь в виду, что сигнатура не включает в себя тип возвращаемого значения, поскольку он не используется в Java при принятии решения о перегрузке.

Перегрузка конструкторов

Как и методы, конструкторы также могут перегружаться, что дает возможность создавать объекты различными способами. В качестве примера рассмотрим следующую программу.

```
// Демонстрация перегрузки конструкторов
class MyClass {
   int x;
```

```
   MyClass() {  ◄──────────────────────── Создание объектов разными способами
      System.out.println("Внутри MyClass().");
      x = 0;
   }

   MyClass(int i) {
      System.out.println("Внутри MyClass(int).");
      x = i;
   }

   MyClass(double d) {
      System.out.println("Внутри MyClass(double).");
      x = (int) d;
   }

   MyClass(int i, int j) {
      System.out.println("Внутри MyClass(int, int).");
      x = i * j;
   }
}

class OverloadConsDemo {
   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass(88);
      MyClass t3 = new MyClass(17.23);
      MyClass t4 = new MyClass(2, 4);

      System.out.println("t1.x: " + t1.x);
      System.out.println("t2.x: " + t2.x);
      System.out.println("t3.x: " + t3.x);
      System.out.println("t4.x: " + t4.x);
   }
}
```

Выполнение этой программы приведет к следующему результату.

```
Внутри MyClass().
Внутри MyClass(int).
Внутри MyClass(double).
Внутри MyClass(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8
```

В данном примере конструктор `MyClass()` перегружается четырежды. Во всех перегруженных версиях этого конструктора объект типа `MyClass` создается по-разному. Конкретный вариант конструктора выбирается на основании параметров, которые указываются при выполнении оператора `new`. Перегружая конструктор класса, вы предоставляете пользователю созданного вами класса свободу в выборе способа конструирования объекта.

Перегрузка конструкторов чаще всего используется для того, чтобы дать возможность инициализировать один объект на основании другого объекта. Рассмотрим в качестве примера следующую программу, в которой класс `Summation` используется для вычисления суммы двух целочисленных значений.

```
// Инициализация одного объекта посредством другого
class Summation {
   int sum;

   // Создание объекта на основе целочисленного значения
   Summation(int num) {   ◄—— Создание одного объекта на основании другого объекта
      sum = 0;
      for(int i=1; i <= num; i++)
         sum += i;
   }

   // Создание одного объекта на основе другого
   Summation(Summation ob) {
      sum = ob.sum;
   }
}

class SumDemo {
   public static void main(String args[]) {
      Summation s1 = new Summation(5);
      Summation s2 = new Summation(s1);

      System.out.println("s1.sum: " + s1.sum);
      System.out.println("s2.sum: " + s2.sum);
   }
}
```

Выполнение этой программы приведет к следующему результату.

```
s1.sum: 15
s2.sum: 15
```

Как следует из приведенного выше примера, использование одного объекта при инициализации другого нередко вполне оправданно. В данном случае при создании объекта `s2` нет необходимости вычислять сумму. Даже если подобная инициализация не повышает быстродействие программы, зачастую удобно иметь конструктор, создающий копию объекта.

Упражнение 6.2 Перегрузка конструктора класса Queue

`QDemo2.java`

В этом упражнении нам предстоит усовершенствовать класс очереди (`Queue`), добавив в него два дополнительных конструктора. В первом из них новая очередь будет конструироваться на основе уже существующей, а во втором начальные значения элементов очереди будут присваиваться при ее конструировании. Как станет ясно в дальнейшем,

добавление этих конструкторов сделает класс Queue более удобным для использования. Поэтапное описание процесса создания соответствующей программы приведено ниже.

1. Создайте новый файл QDemo2.java и скопируйте в него код класса Queue, созданный в упражнении 6.1.
2. Добавьте в этот класс приведенный ниже конструктор, который будет создавать одну очередь на основе другой.

```
// Конструктор, создающий один объект
// типа Queue на основе другого
Queue(Queue ob) {
   putloc = ob.putloc;
   getloc = ob.getloc;
   q = new char[ob.q.length];

   // Копирование элементов очереди
   for(int i=getloc+1; i <= putloc; i++)
      q[i] = ob.q[i];
}
```

Внимательно проанализируем работу этого конструктора. Сначала переменные putloc и getloc инициализируются значениями, содержащимися в объекте ob, который передается в качестве параметра. Затем организуется новый массив для хранения элементов очереди, которые далее копируются из объекта ob в этот массив. Вновь созданная копия очереди будет идентична оригиналу, хотя они и являются совершенно отдельными объектами.

3. Добавьте в данный класс конструктор, инициализирующий очередь данными из символьного массива, как показано ниже.

```
// Конструирование и инициализация объекта типа Queue
Queue(char a[]) {
   putloc = 0;
   getloc = 0;
   q = new char[a.length+1];

   for(int i = 0; i < a.length; i++) put(a[i]);
}
```

В этом конструкторе создается достаточно большая очередь для хранения символов из массива a. В силу особенностей алгоритма, реализующего очередь, длина очереди должна быть на один элемент больше, чем длина исходного массива.

4. Ниже приведен завершенный код видоизмененного класса Queue, а также код класса QDemo2, демонстрирующего организацию очереди для хранения символов.

```
// Класс, реализующий очередь для хранения символов
class Queue {
   private char q[]; // массив для хранения элементов очереди
```

```
    private int putloc, getloc;  // индексы для вставки и
                                 // извлечения элементов
                                 // очереди

    // Создание пустой очереди заданного размера
    Queue(int size) {
       q = new char[size+1]; // выделение памяти для очереди
       putloc = getloc = 0;
    }

    // Создание очереди на основе имеющегося объекта Queue
    Queue(Queue ob) {
       putloc = ob.putloc;
       getloc = ob.getloc;
       q = new char[ob.q.length];

       // Копирование элементов в очередь
       for(int i=getloc+1; i <= putloc; i++)
          q[i] = ob.q[i];
    }

    // Создание очереди на основе массива исходных значений
    Queue(char a[]) {
       putloc = 0;
       getloc = 0;
       q = new char[a.length+1];

       for(int i = 0; i < a.length; i++) put(a[i]);
    }

    // Помещение символа в очередь
    void put(char ch) {
       if(putloc==q.length-1) {
          System.out.println(" - Очередь заполнена");
          return;
       }

       q[putloc++] = ch;
    }

    // Извлечение символа из очереди
    char get() {
       if(getloc == putloc) {
          System.out.println(" - Очередь пуста");
          return (char) 0;
       }

       return q[getloc++];
    }
}
```

```
// Демонстрация использования класса Queue
class QDemo2 {
   public static void main(String args[]) {
      // Создание пустой очереди для хранения 10 элементов
      Queue q1 = new Queue(10);

      char name[] = {'T', 'o', 'm'};
      // Создание очереди на основе массива
      Queue q2 = new Queue(name);

      char ch;
      int i;

      // Помещение ряда символов в очередь q1
      for(i=0; i < 10; i++)
         q1.put((char) ('A' + i));

      // Создание одной очереди на основе другой
      Queue q3 = new Queue(q1);

      // Отображение очередей
      System.out.print("Содержимое q1: ");
      for(i=0; i < 10; i++) {
         ch = q1.get();
         System.out.print(ch);
      }

      System.out.println("\n");

      System.out.print("Содержимое q2: ");
      for(i=0; i < 3; i++) {
         ch = q2.get();
         System.out.print(ch);
      }

      System.out.println("\n");

      System.out.print("Содержимое q3: ");
      for(i=0; i < 10; i++) {
         ch = q3.get();
         System.out.print(ch);
      }
   }
}
```

5. Результат выполнения данной программы выглядит следующим образом.

```
Содержимое q1: ABCDEFGHIJ

Содержимое q2: Tom

Содержимое q3: ABCDEFGHIJ
```

Рекурсия

В Java допускается, чтобы метод вызывал сам себя. Этот процесс называется *рекурсией*, а метод, вызывающий сам себя, — *рекурсивным*. Вообще говоря, рекурсия представляет собой процесс, в ходе которого некая сущность определяет себя же. В этом отношении она чем-то напоминает циклическое определение. Рекурсивный метод отличается в основном тем, что он содержит инструкцию, в которой этот метод вызывает сам себя. Рекурсия является эффективным механизмом управления программой.

Классическим примером рекурсии служит вычисление факториала числа. *Факториал* числа N — это произведение всех целых чисел от 1 до *N*. Например, факториал числа 3 равен 1×2×3, или 6. В приведенном ниже примере программы демонстрируется рекурсивный способ вычисления факториала числа. Для сравнения в эту программу включен также нерекурсивный вариант вычисления факториала.

```
// Простой пример рекурсии
class Factorial {
   // Рекурсивный метод
   int factR(int n) {
      int result;

      if(n==1) return 1;
      result = factR(n-1) * n;
      return result;
   }                   ← Рекурсивный вызов метода factR()

   // Вариант программы, вычисляющий факториал
   // итеративным способом
   int factI(int n) {
      int t, result;

      result = 1;
      for(t=1; t <= n; t++) result *= t;
      return result;
   }
}

class Recursion {
   public static void main(String args[]) {
      Factorial f = new Factorial();

      System.out.println("Вычисление рекурсивным методом");
      System.out.println("Факториал 3 равен " + f.factR(3));
      System.out.println("Факториал 4 равен " + f.factR(4));
      System.out.println("Факториал 5 равен " + f.factR(5));
      System.out.println();
```

```
        System.out.println("Вычисление итеративным методом");
        System.out.println("Факториал 3 равен " + f.factI(3));
        System.out.println("Факториал 4 равен " + f.factI(4));
        System.out.println("Факториал 5 равен " + f.factI(5));
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Вычисление рекурсивным методом
Факториал 3 равен 6
Факториал 4 равен 24
Факториал 5 равен 120

Вычисление итеративным методом
Факториал 3 равен 6
Факториал 4 равен 24
Факториал 5 равен 120
```

Действия нерекурсивного (итеративного) метода `factI()` не требуют особых пояснений. В нем используется цикл, в котором числа, начиная с 1, последовательно умножаются друг на друга, постепенно образуя произведение, дающее факториал.

Рекурсивный метод `factR()` действует в соответствии с более сложной схемой. Когда метод `factR()` вызывается с аргументом, равным 1, он возвращает 1, в противном случае — произведение, определяемое из выражения `factR(n-1)*n`. Для вычисления этого выражения вызывается метод `factR()` с аргументом `n-1`. Этот процесс повторяется до тех пор, пока значение переменной `n` не окажется равным 1, после чего из предыдущих вызовов данного метода начнут возвращаться полученные значения. Например, при вычислении факториала 2 первый вызов метода `factR()` повлечет за собой второй вызов того же самого метода, но с аргументом 1. В результате метод вернет значение 1, которое затем умножается на 2 (т.е. исходное значение переменной `n`). В результате всех этих вычислений будет получен факториал, равный 2. По желанию в теле метода `factR()` можно добавить вызовы `println()`, чтобы сообщать, на каком именно уровне осуществляется очередной вызов, а также отображать промежуточные результаты вычислений.

Когда метод вызывает сам себя, в системном стеке распределяется память для новых локальных переменных и параметров, и код метода выполняется с этими новыми переменными и параметрами с самого начала. При рекурсивном вызове метода не создается его новая копия, но лишь используются его новые аргументы. А при возврате из каждого рекурсивного вызова старые локальные переменные и параметры извлекаются из стека, и выполнение возобновляется с точки вызова в методе. Рекурсивные методы можно сравнить по принципу действия с постепенно сжимающейся и затем распрямляющейся пружиной.

Рекурсивные варианты многих процедур могут выполняться немного медленнее, чем их итерационные эквиваленты, из-за дополнительных затрат

системных ресурсов на неоднократные вызовы метода. Если же таких вызовов окажется слишком много, то в конечном итоге системный стек может быть переполнен. А поскольку параметры и локальные переменные рекурсивного метода хранятся в системном стеке и при каждом очередном вызове этого метода создается их новая копия, то в какой-то момент стек может оказаться исчерпанным. Если возникнет подобная ситуация, исполняющая среда Java сгенерирует исключение. Но в большинстве случаев об этом не стоит особо беспокоиться. Как правило, переполнение системного стека происходит тогда, когда рекурсивный метод выходит из-под контроля.

Главное преимущество рекурсии заключается в том, что она позволяет реализовать некоторые алгоритмы яснее и проще, чем итерационным способом. Например, алгоритм быстрой сортировки довольно трудно реализовать итерационным способом. А некоторые задачи, например, искусственного интеллекта, очевидно, требуют именно рекурсивного решения. При написании рекурсивных методов следует указать в соответствующем месте условную инструкцию, например `if`, чтобы организовать возврат из метода без рекурсии. В противном случае возврат из вызванного однажды рекурсивного метода может вообще не произойти. Подобного рода ошибка весьма характерна для реализации рекурсии в практике программирования. Поэтому рекомендуется пользоваться инструкциями, содержащими вызовы метода `println()`, чтобы следить за происходящим в рекурсивном методе и прервать его выполнение, если в нем обнаружится ошибка.

Применение ключевого слова `static`

Иногда требуется определить такой член класса, который будет использоваться независимо от каких бы то ни было объектов этого класса. Как правило, доступ к члену класса организуется посредством объекта этого класса, но в то же время можно создать член класса для самостоятельного применения без ссылки на конкретный объект. Для того чтобы создать такой член класса, достаточно указать в самом начале его объявления ключевое слово `static`. Если член класса объявляется как `static`, он становится доступным до создания каких-либо объектов своего класса и без ссылки на какой-либо объект. С помощью ключевого слова `static` можно объявлять как переменные, так и методы. Такие члены и методы называются статическими. Наиболее характерным примером члена типа `static` служит метод `main()`, который объявляется таковым потому, что он должен вызываться виртуальной машиной Java в самом начале выполняемой программы. Для того чтобы воспользоваться членом типа `static` за пределами класса, достаточно дополнить имя данного члена именем класса, используя точечную нотацию. Но создавать объект для этого не нужно. В действительности член типа `static` оказывается доступным не по ссылке на объект, а по имени своего класса. Так, если требуется присвоить значение 10

переменной `count` типа `static`, являющейся членом класса `Timer`, то для этой цели можно воспользоваться следующей строкой кода:

```
Timer.count = 10;
```

Эта форма записи подобна той, что используется для доступа к обычным переменным экземпляра посредством объекта, но в ней указывается имя класса, а не объекта. Аналогичным образом вызываются методы типа `static`.

Переменные, объявляемые как `static`, по сути являются глобальными. В силу этого при создании объектов данного класса копии статических переменных в них не создаются. Вместо этого все экземпляры класса совместно пользуются одной и той же статической переменной. Ниже приведен пример программы, демонстрирующий различия между статическими и обычными переменными экземпляра.

```
// Применение статической переменной
class StaticDemo {
   int x; // обычная переменная экземпляра
   static int y; // статическая переменная   ◄── Все объекты используют одну и ту
                                                 же копию статической переменной
   // Возврат суммы значений переменной экземпляра x и
   // статической переменной y
   int sum() {
      return x + y;
   }
}

class SDemo {
   public static void main(String args[]) {
      StaticDemo ob1 = new StaticDemo();
      StaticDemo ob2 = new StaticDemo();

      // У каждого объекта имеется своя копия
      // переменной экземпляра
      ob1.x = 10;
      ob2.x = 20;
      System.out.println("Разумеется, ob1.x и ob2.x " +
                         "независимы");
      System.out.println("ob1.x: " + ob1.x +
                         "\nob2.x: " + ob2.x);
      System.out.println();

      // Все объекты совместно используют одну общую
      // копию статической переменной
      System.out.println("Статическая переменная y - общая");
      StaticDemo.y = 19;
      System.out.println("Присвоить StaticDemo.y значение 19");

      System.out.println("ob1.sum(): " + ob1.sum());
      System.out.println("ob2.sum(): " + ob2.sum());
      System.out.println();
```

```
        StaticDemo.y = 100;
        System.out.println("Изменить значение StaticDemo.y на 100");

        System.out.println("ob1.sum(): " + ob1.sum());
        System.out.println("ob2.sum(): " + ob2.sum());
        System.out.println(); }
}
```

Выполнение этой программы дает следующий результат.

```
Разумеется, ob1.x and ob2.x независимы
ob1.x: 10
ob2.x: 20

Статическая переменная y - общая
Присвоить StaticDemo.y значение 19
ob1.sum(): 29
ob2.sum(): 39

Изменить значение StaticDemo.y на 100
ob1.sum(): 110
ob2.sum(): 120
```

Нетрудно заметить, что статическая переменная y используется как объектом ob1, так и объектом ob2. Изменения в ней оказывают влияние на весь класс, а не только на его экземпляр.

Метод типа static отличается от обычного метода тем, что его можно вызывать по имени его класса, не создавая экземпляр объекта этого класса. Пример такого вызова уже приводился ранее. Это был метод sqrt() типа static, относящийся к классу Math из стандартной библиотеки классов Java. Ниже приведен пример программы, в которой объявляется статическая переменная и создается метод типа static.

```
// Применение статического метода
class StaticMeth {
   static int val = 1024; // статическая переменная

   // Статический метод
   static int valDiv2() {
      return val/2;
   }
}

class SDemo2 {
   public static void main(String args[]) {

      System.out.println("Значение val: " + StaticMeth.val);
      System.out.println("StaticMeth.valDiv2(): " +
                         StaticMeth.valDiv2());
```

```
        StaticMeth.val = 4;
        System.out.println("Значение val: " + StaticMeth.val);
        System.out.println("StaticMeth.valDiv2(): " +
                           StaticMeth.valDiv2());
    }
}
```

Выполнение этой программы дает следующий результат.

```
Значение val: 1024
StaticMeth.valDiv2(): 512
Значение val: 4
StaticMeth.valDiv2(): 2
```

На применение методов типа `static` накладывается ряд следующих ограничений:

- в методе типа `static` допускается непосредственный вызов только других методов типа `static`;
- для метода типа `static` непосредственно доступными оказываются только другие данные типа `static`, определенные в его классе;
- в методе типа `static` должна отсутствовать ссылка `this`.

В приведенном ниже классе код статического метода `valDivDenom()` создан некорректно.

```
class StaticError {
    int denom = 3;          // обычная переменная экземпляра
    static int val = 1024; // статическая переменная

    // Ошибка! К нестатическим переменным нельзя обращаться
    // из статического метода.
    static int valDivDenom() {
        return val/denom; // не пройдет компиляцию!
    }
}
```

В данном примере `denom` является обычной переменной экземпляра, к которой нельзя обращаться из статического метода.

Статические блоки

Иногда для подготовки к созданию объектов в классе должны быть выполнены некоторые инициализирующие действия. В частности, может возникнуть потребность установить соединение с удаленным сетевым узлом или задать значения некоторых статических переменных перед тем, как воспользоваться статическими методами класса. Для решения подобных задач в Java предусмотрены статические блоки (`static`). Статический блок выполняется при первой загрузке класса, еще до того, как класс будет использован для каких-нибудь других целей. Ниже приведен пример применения статического блока.

```
// Применение статического блока
class StaticBlock {
   static double rootOf2;
   static double rootOf3;

   static {  ◄—————————————————————— Этот блок выполняется при загрузке класса
      System.out.println("Внутри статического блока");
      rootOf2 = Math.sqrt(2.0);
      rootOf3 = Math.sqrt(3.0);
   }

   StaticBlock(String msg) {
      System.out.println(msg);
   }
}

class SDemo3 {
   public static void main(String args[]) {
      StaticBlock ob = new StaticBlock("Внутри конструктора");

      System.out.println("Корень квадратный из 2 равен " +
                         StaticBlock.rootOf2);
      System.out.println("Корень квадратный из 3 равен " +
                         StaticBlock.rootOf3);
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
Внутри статического блока
Внутри конструктора
Корень квадратный из 2 равен 1.4142135623730951
Корень квадратный из 3 равен 1.7320508075688772
```

Как видите, статический блок выполняется еще до того, как будет создан какой-либо объект.

Упражнение 6.3 Быстрая сортировка

QSDemo.java

В главе 5 был рассмотрен простой способ так называемой пузырьковой сортировки. Там же было вкратце упомянуто о том, что существуют лучшие способы сортировки. В этом упражнении нам предстоит реализовать один из самых эффективных способов: быструю сортировку. Алгоритм быстрой сортировки был разработан Чарльзом Хоаром и назван его именем. На сегодняшний день это самый лучший универсальный алгоритм сортировки. Он не был продемонстрирован в главе 5 лишь потому, что реализовать быструю сортировку лучше всего с помощью рекурсии. В данном упражнении будет создана программа для сортировки символьного массива, но демонстрируемый подход может быть применен к сортировке любых объектов.

Быстрая сортировка опирается на принцип разделения. Сначала из массива выбирается один опорный элемент (так называемый *компаранд*), и массив делится на две части. Элементы, меньшие опорного, помещаются в одну часть массива, а большие или равные опорному — в другую часть. Затем процесс рекурсивно повторяется для каждой оставшейся части до тех пор, пока массив не окажется отсортированным. Допустим, имеется массив, содержащий последовательность символов fedacb, а в качестве опорного выбран символ d. На первом проходе массив будет частично упорядочен следующим образом:

Исходные данные	f e d a c b
Проход 1	b c a d e f

Далее этот процесс повторяется для каждой части: bca и def. Как видите, процесс рекурсивен по своей сути, и действительно, наиболее эффективной реализацией быстрой сортировки является рекурсивная сортировка.

Опорный элемент можно выбрать двумя способами: случайным образом или путем вычисления среднего значения части элементов массива. Эффективность сортировки будет оптимальной в том случае, когда опорный элемент выбирается как раз посредине диапазона значений элементов, содержащихся в массиве, но зачастую выбрать такое значение непросто. Если же опорный элемент выбирается случайным образом, то вполне возможно, что он окажется на краю диапазона. Но и в этом случае алгоритм быстрой сортировки будет действовать корректно. В том варианте быстрой сортировки, который реализуется в данном упражнении, в качестве опорного выбирается элемент, находящийся посередине массива. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл QSDemo.java.
2. Создайте класс Quicksort, код которого приведен ниже.

```
// Упражнение 6.3. Простая версия класса Quicksort,
// реализующего быструю сортировку
class Quicksort {

   // Вызов фактического метода быстрой сортировки
   static void qsort(char items[]) {
      qs(items, 0, items.length-1);
   }

   // Рекурсивная версия метода быстрой сортировки символов
   private static void qs(char items[], int left, int right)
   {
      int i, j;
      char x, y;

      i = left; j = right;
      x = items[(left+right)/2];
```

```
        do {
            while((items[i] < x) && (i < right)) i++;
            while((x < items[j]) && (j > left)) j--;

            if(i <= j) {
                y = items[i];
                items[i] = items[j];
                items[j] = y;
                i++; j--;
            }
        } while(i <= j);

        if(left < j) qs(items, left, j);
        if(i < right) qs(items, i, right);
    }
}
```

С целью упрощения интерфейса в классе `Quicksort` предоставляется метод `qsort()`, из которого вызывается метод `qs()`, фактически выполняющий сортировку. Подобный подход позволяет выполнять сортировку, передавая методу лишь имя массива и не осуществляя первоначальное разделение. А поскольку метод `qs()` используется только в классе, он определяется как `private`.

3. Для того чтобы запустить сортировку, достаточно вызвать метод `Quicksort.qsort()`. Этот метод определен как `static`, и поэтому для его вызова достаточно указать имя класса, а создавать объект не обязательно. По завершении работы этого метода массив будет отсортирован. Данная версия программы работает только с символьными массивами, но ее можно адаптировать для сортировки массивов любого типа.
4. Ниже приведен полный исходный код программы, демонстрирующей применение класса `Quicksort`.

```
// Упражнение 6.3. Простая версия класса Quicksort,
// реализующего быструю сортировку
class Quicksort {

    // Вызов фактического метода быстрой сортировки
    static void qsort(char items[]) {
        qs(items, 0, items.length-1);
    }

    // Рекурсивная версия метода быстрой сортировки символов
    private static void qs(char items[], int left, int right)
    {
        int i, j;
        char x, y;

        i = left; j = right;
        x = items[(left+right)/2];
```

```
        do {
            while((items[i] < x) && (i < right)) i++;
            while((x < items[j]) && (j > left)) j--;

            if(i <= j) {
                y = items[i];
                items[i] = items[j];
                items[j] = y;
                i++; j--;
            }
        } while(i <= j);

        if(left < j) qs(items, left, j);
        if(i < right) qs(items, i, right);
    }
}

class QSDemo {
    public static void main(String args[]) {
        char a[] = { 'd', 'x', 'a', 'r', 'p', 'j', 'i' };
        int i;

        System.out.print("Исходный массив: ");
        for(i=0; i < a.length; i++)
            System.out.print(a[i]);

        System.out.println();

        // Сортировка массива
        Quicksort.qsort(a);

        System.out.print("Отсортированный массив: ");
        for(i=0; i < a.length; i++)
            System.out.print(a[i]);
    }
}
```

Вложенные и внутренние классы

В Java определены вложенные классы. *Вложенным* называется такой класс, который объявляется в другом классе. Вложенные классы не относятся к базовым языковым средствам Java. Они даже не поддерживались до появления версии Java 1.1, хотя с тех пор часто применяются в реальных программах, и поэтому о них нужно знать.

Вложенный класс не может существовать независимо от класса, в который он вложен. Следовательно, область действия вложенного класса ограничена его внешним классом. Если вложенный класс объявлен в пределах области действия внешнего класса, то он становится членом последнего. Имеется также возможность объявить вложенный класс, который станет локальным в пределах блока.

Существуют два типа вложенных классов. Одни вложенные классы объявляются с помощью модификатора доступа `static`, а другие — без него. В этой книге будет рассматриваться только нестатический вариант вложенных классов. Классы такого типа называются ***внутренними***. Внутренний класс имеет доступ ко всем переменным и методам внешнего класса, в который он вложен, и может обращаться к ним непосредственно, как и все остальные нестатические члены внешнего класса.

Иногда внутренний класс используется для предоставления ряда услуг внешнему классу, в котором он содержится. Ниже приведен пример применения внутреннего класса для вычисления различных значений, которые используются включающим его классом.

```
// Применение внутреннего класса
class Outer {
   int nums[];

   Outer(int n[]) {
      nums = n;
   }

   void Analyze() {
      Inner inOb = new Inner();

      System.out.println("Минимум: " + inOb.min());
      System.out.println("Максимум: " + inOb.max());
      System.out.println("Среднее: " + inOb.avg());
   }

   // Внутренний класс
   class Inner { ◄——— Внутренний класс
      int min() {
         int m = nums[0];

         for(int i=1; i < nums.length; i++)
            if(nums[i] < m) m = nums[i];
         return m;
      }

      int max() {
         int m = nums[0];
         for(int i=1; i < nums.length; i++)
            if(nums[i] > m) m = nums[i];

         return m;
      }

      int avg() {
         int a = 0;
         for(int i=0; i < nums.length; i++)
            a += nums[i];
```

```
            return a / nums.length;
        }
    }
}

class NestedClassDemo {
    public static void main(String args[]) {
        int x[] = { 3, 2, 1, 5, 6, 9, 7, 8 };
        Outer outOb = new Outer(x);

        outOb.Analyze();
    }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
Минимум: 1
Максимум: 9
Среднее: 5
```

В данном примере внутренний класс `Inner` обрабатывает массив `nums`, являющийся членом класса `Outer`. Вложенный класс имеет доступ к членам охватывающего класса и поэтому может непосредственно обращаться к массиву `nums`. А вот обратное не справедливо. Так, например, метод `analyze()` не может непосредственно вызвать метод `min()`, не создав объект типа `Inner`.

Как уже упоминалось, класс можно вложить в области действия блока. В итоге получается локальный класс, недоступный за пределами блока. В следующем примере программы мы преобразуем класс `ShowBits`, созданный в упражнении 5.3, таким образом, чтобы он стал локальным.

```
// Применение класса ShowBits в качестве локального
class LocalClassDemo {
    public static void main(String args[]) {

        // Внутренняя версия класса ShowBits
        class ShowBits { ◄──────────── Локальный класс, вложенный в метод
            int numbits;

            ShowBits(int n) {
                numbits = n;
            }

            void show(long val) {
                long mask = 1;

                // Сдвиг влево для установки единицы в нужной позиции
                mask <<= numbits-1;

                int spacer = 0;
                for(; mask != 0; mask >>>= 1) {
                    if((val & mask) != 0) System.out.print("1");
                    else System.out.print("0");
```

```
                spacer++;
                if((spacer % 8) == 0) {
                    System.out.print(" ");
                    spacer = 0;
                }
            }
            System.out.println();
        }
    }

    for(byte b = 0; b < 10; b++) {
        ShowBits byteval = new ShowBits(8);

        System.out.print(b + " в двоичном представлении: ");
        byteval.show(b);
    }
  }
}
```

Выполнение этой программы дает следующий результат.

```
0 в двоичном представлении: 00000000
1 в двоичном представлении: 00000001
2 в двоичном представлении: 00000010
3 в двоичном представлении: 00000011
4 в двоичном представлении: 00000100
5 в двоичном представлении: 00000101
6 в двоичном представлении: 00000110
7 в двоичном представлении: 00000111
8 в двоичном представлении: 00001000
9 в двоичном представлении: 00001001
```

В данном примере класс ShowBits недоступен за пределами метода main(), а следовательно, попытка получить доступ к нему из любого метода, кроме main(), приведет к ошибке.

И последнее замечание: внутренний класс может быть безымянным. Экземпляр ***анонимного внутреннего класса*** создается при объявлении класса с помощью оператора new. Анонимные внутренние классы будут подробнее рассмотрены в главе 16.

Спросим у эксперта

ВОПРОС. Чем статический вложенный класс отличается от нестатического?

ОТВЕТ. Статический вложенный класс объявляется с помощью модификатора static. Являясь статическим, он может непосредственно обращаться к любому статическому члену своего внешнего класса. Другие члены внешнего класса доступны ему посредством ссылки на объект.

Переменное число аргументов

Иногда оказываются полезными методы, способные принимать переменное число аргументов. Например, методу, устанавливающему соединение с Интернетом, могут понадобиться имя и пароль пользователя, имя файла, протокол и другие параметры. Если при вызове метода некоторые из этих данных опущены, то должны использоваться значения по умолчанию. В подобных ситуациях было бы удобнее передавать только те аргументы, для которых заданные по умолчанию значения неприменимы. Для этого требуется метод, который мог бы принимать аргументы, количество которых заранее неизвестно.

Ранее для поддержки списков аргументов переменной длины применялись два способа, ни один из которых не был особенно удобен. Во-первых, если максимально возможное количество аргументов было невелико и заранее известно, то можно было создавать перегруженные версии метода — по одной для каждого способа его вызова. Очевидно, что такой подход применим лишь в отдельных случаях. И во-вторых, если таких версий требовалось создавать слишком много или их максимальное количество было неопределенным, то применялся второй подход: параметры помещались в массив, а затем этот массив передавался методу. У каждого из этих способов имеются свои недостатки, и со временем стало ясно, что для преодоления описанной проблемы следует искать другие решения.

Такое решение было предложено в JDK 5. Новое средство, которое позволило избавиться от явного формирования массива аргументов перед вызовом метода, получило название *varargs* (от "variable-length arguments" — список аргументов переменной длины). Соответствующие методы называют ***методами с переменным числом аргументов*** (другое название — ***методы переменной арности***). В методах этого типа список параметров имеет не фиксированную, а переменную длину, что обеспечивает дополнительную гибкость, позволяя методам иметь произвольное число аргументов.

Использование методов с переменным числом аргументов

Списки аргументов переменной длины обозначаются символом многоточия (`...`). Ниже приведен пример метода `vaTest()`, имеющего переменное число аргументов (в том числе и нулевое).

Объявление метода со списком аргументов переменной длины

```
// Метод vaTest() с переменным числом аргументов
static void vaTest(int ... v) { ◄────
   System.out.println("Число аргументов: " + v.length);
   System.out.println("Содержимое: ");

   for(int i=0; i < v.length; i++)
      System.out.println(" arg " + i + ": " + v[i]);

   System.out.println();
}
```

Обратите внимание на синтаксис объявления параметра v:

```
int ... v
```

Это объявление сообщает компилятору, что метод vaTest() может вызываться с указанием произвольного количества аргументов, в том числе и вовсе без них. Более того, оно означает неявное объявление аргумента v как массива типа int[]. Таким образом, в теле метода vaTest() доступ к параметру v осуществляется с помощью обычного синтаксиса обращения к массивам.

Ниже приведен полный исходный код примера программы, которая демонстрирует использование метода vaTest().

```
// Демонстрация использования метода
// с переменным числом аргументов
class VarArgs {

   // Метод vaTest() допускает переменное число аргументов
   static void vaTest(int ... v) {
      System.out.println("Количество аргументов: " + v.length);
      System.out.println("Содержимое: ");

      for(int i=0; i < v.length; i++)
         System.out.println(" arg " + i + ": " + v[i]);

      System.out.println();
   }

   public static void main(String args[])
   {
      // Метод vaTest() может вызываться с
      // переменным числом аргументов
      vaTest(10);       // 1 аргумент
      vaTest(1, 2, 3);  // 3 аргумента
      vaTest();         // без аргументов
  }
}
```

Вызовы метода с указанием различного числа аргументов

Выполнение этой программы дает следующий результат.

```
Количество аргументов: 1
Содержимое:
arg 0: 10

Количество аргументов: 3
Содержимое:
arg 0: 1
arg 1: 2
arg 2: 3

Количество аргументов: 0
Содержимое:
```

В приведенной выше программе обращает на себя внимание следующее. Во-первых, как пояснялось выше, обращение к параметру v в методе vaTest() осуществляется как к массиву. Дело в том, что он действительно является массивом (и, таким образом, может иметь переменную длину). Многоточие в объявлении этого метода указывает компилятору на использование переменного числа аргументов, а также на необходимость поместить их в массив v. Во-вторых, в методе main() имеются вызовы метода vaTest() с использованием различного числа аргументов, в том числе и без указания аргумента. Указываемые аргументы автоматически помещаются в массив v. Если же аргументы не указаны, длина этого массива будет равна нулю.

Помимо списка параметров переменной длины, в объявлении метода могут указываться и обычные параметры, но при одном условии: массив параметров переменной длины должен быть указан последним. Например, приведенное ниже объявление метода является вполне допустимым:

```
int doIt(int a, int b, double c, int ... vals) {
```

В этом случае первым трем аргументам, указанным при вызове метода doIt(), будут соответствовать первые три параметра в объявлении метода, тогда как остальные аргументы будут браться из массива vals.

Ниже приведен переработанный вариант метода vaTest(), в котором метод получает как обычные аргументы, так и массив аргументов переменной длины.

```
// Использование массива аргументов переменной длины
// наряду с обычными аргументами
class VarArgs2 {

   // Здесь msg - обычный параметр,
   // а v - массив параметров переменной длины
   static void vaTest(String msg, int ... v) {   ◄── "Обычный" параметр и параметр в виде массива переменной длины
      System.out.println(msg + v.length);
      System.out.println("Содержимое: ");

      for(int i=0; i < v.length; i++)
         System.out.println(" arg " + i + ": " + v[i]);

      System.out.println();
   }

   public static void main(String args[])
   {
      vaTest("Один аргумент в массиве: ", 10);
      vaTest("Три аргумента в массиве: ", 1, 2, 3);
      vaTest("Отсутствуют аргументы в виде массива: ");
   }
}
```

Выполнение этого фрагмента кода дает следующий результат.

```
Один аргумент в массиве: 1
Содержимое:
arg 0: 10

Три аргумента в массиве: 3
Содержимое:
arg 0: 1
arg 1: 2
arg 2: 3

Отсутствуют аргументы в массиве: 0
Содержимое:
```

Помните о том, что список параметров переменной длины должен указываться последним. Например, следующее объявление метода недопустимо.

```
int doIt(int a, int b, double c, int ... vals,
        boolean stopFlag) { // Ошибка!
```

В данном примере сделана попытка указать обычный параметр после списка параметров переменной длины.

Существует еще одно ограничение, которое следует соблюдать: список параметров переменной длины можно указать в методе только один раз. Например, приведенное ниже объявление метода недопустимо.

```
int doIt(int a, int b, double c, int ... vals,
        double ... morevals) { // Ошибка!
```

Ошибкой в данном случае является попытка указать два разных списка параметров переменной длины.

Перегрузка методов с переменным числом аргументов

Методы, имеющие переменное число аргументов, можно перегружать. Например, в следующей программе представлены три перегруженные версии метода `vaTest()`.

```
// Перегрузка метода с переменным числом аргументов
class VarArgs3 {
                                  Первая версия метода vaTest()
   static void vaTest(int ... v) { ◄
      System.out.println("vaTest(int ...): " +
                         "Количество аргументов: " + v.length);
      System.out.println("Содержимое: ");

      for(int i=0; i < v.length; i++)
         System.out.println(" arg " + i + ": " + v[i]);

      System.out.println();
   }
                                     Вторая версия метода vaTest()
   static void vaTest(boolean ... v) { ◄
      System.out.println("vaTest(boolean ...): " +
```

```
                        "Количество аргументов: " + v.length);
      System.out.println("Содержимое: ");

      for(int i=0; i < v.length; i++)
         System.out.println(" arg " + i + ": " + v[i]);

      System.out.println();
   }
                                                  Третья версия метода vaTest()
   static void vaTest(String msg, int ... v) { ◄──┘
      System.out.println("vaTest(String, int ...): " +
                         msg + v.length);
      System.out.println("Содержимое: ");

      for(int i=0; i < v.length; i++)
         System.out.println(" arg " + i + ": " + v[i]);

      System.out.println();
   }

   public static void main(String args[])
   {
      vaTest(1, 2, 3);
      vaTest("Тестирование: ", 10, 20);
      vaTest(true, false, false);
   }
}
```

Выполнение этой программы дает следующие результаты.

```
vaTest(int ...): Количество аргументов: 3
Содержимое:
   arg 0: 1
   arg 1: 2
   arg 2: 3

vaTest(String, int ...): Тестирование: 2
Содержимое:
   arg 0: 10
   arg 1: 20

vaTest(boolean ...): Количество аргументов: 3
Содержимое:
   arg 0: true
   arg 1: false
   arg 2: false
```

В этой программе продемонстрированы два способа перегрузки методов с переменным числом аргументов. Во-первых, перегруженные версии методов могут различаться типом параметра, содержащего переменное количество аргументов. По этому принципу перегружены версии метода `vaTest(int ...)` и `vaTest(boolean ...)`. Вспомните, что многоточие говорит о том, что

соответствующий аргумент должен рассматриваться как массив указанного типа. Поэтому, в полной аналогии с тем, как обычные методы можно перегружать за счет использования различных типов параметра, соответствующего массиву, методы с переменным числом аргументов можно перегружать, используя различные типы параметра varargs. На основании этого различия и будет определяться версия, подлежащая вызову.

Второй способ перегрузки методов с переменным числом аргументов состоит в добавлении одного или нескольких обычных аргументов. Он реализован в версии метода vaTest(String, int ...). В этом случае исполняющая среда Java использует для выбора нужной версии метода как число параметров, так и их типы.

Переменное число аргументов и неоднозначность

Перегрузка методов, имеющих список параметров переменной длины, может приводить к возникновению непредвиденных ошибок. Причиной их появления является неоднозначность, которая может возникать при вызове перегруженного метода с переменным числом аргументов. В качестве примера рассмотрим следующую программу.

```
// Перегрузка метода с переменным числом аргументов
// и неоднозначность в выборе перегруженной версии.
//
// В этой программе имеется ошибка, и
// поэтому она не будет компилироваться.
class VarArgs4 {

   // Использование списка аргументов переменной длины типа int
   static void vaTest(int ... v) {  ◄——— Аргументы переменной длины типа int
      // ...
   }

   // Использование списка аргументов переменной длины типа boolean
   static void vaTest(boolean ... v) {  ◄——— Аргументы переменной длины
      // ...                                  типа boolean
   }

   public static void main(String args[])
   {
      vaTest(1, 2, 3); // OK
      vaTest(true, false, false); // OK

      vaTest(); // Ошибка: неоднозначность вызова!  ◄——— Неопределенность!
   }
}
```

В этой программе перегрузка метода vaTest() выполнена совершенно правильно, но программа не будет скомпилирована из-за наличия следующего вызова:

```
vaTest(); // Ошибка: неоднозначность вызова!
```

Вспомните, что переменное количество аргументов допускает и полное их отсутствие, так что в этом отношении все нормально. Однако приведенный выше вызов не может быть однозначно интерпретирован, поскольку ему соответствуют обе перегруженные версии метода: `vaTest(int ...)` и `vaTest(boolean ...)`.

Рассмотрим еще один пример возникновения неоднозначности при обращении к методу. Из двух приведенных ниже версий метода `vaTest()` компилятор не сможет однозначно выбрать требуемую, хотя, казалось бы, одна из них явно отличается от другой наличием дополнительного обычного аргумента.

```
static void vaTest(int ... v) { // ...

static void vaTest(int n, int ... v) { // ...
```

И тем не менее компилятор не сможет определить, какую именно из этих двух версий необходимо использовать для следующего вызова:

```
vaTest(1)
```

Действительно, здесь совершенно неясно, каким образом следует интерпретировать эту строку кода: как вызов метода `vaTest(int ...)` с одним аргументом в виде списка параметров переменной длины или как вызов метода `vaTest(int, int ...)` с отсутствующим списком аргументов переменной длины? Таким образом, в подобных ситуациях также возникает неоднозначность.

В силу описанных причин в ряде случаев имеет смысл отказаться от перегрузки, а для различения методов присвоить им разные имена. Кроме того, в некоторых случаях возникновение ошибок неоднозначности может указывать на то, что при проектировании программы были допущены просчеты, которые можно исправить, более тщательно продумав структуру программы.

Вопросы и упражнения для самопроверки

1. Предположим, имеется следующий фрагмент кода:

   ```
   class X {
      private int count;
   ```

   Исходя из этого, допустим ли следующий код?

   ```
   class Y {
      public static void main(String args[]) {
         X ob = new X();

         ob.count = 10;
   ```

2. Модификатор доступа должен ________________ объявлению члена класса.

3. Помимо очереди, в программах часто используется структура данных, которая называется стеком. Обращение к стеку осуществляется по принципу

"первым пришел — последним обслужен". Стек можно сравнить со стопкой тарелок, стоящих на столе. Последней берется тарелка, поставленная на стол первой. Создайте класс Stack, реализующий стек для хранения символов. Используйте методы push() и pop() для манипулирования содержимым стека. Пользователь класса Stack должен иметь возможность задавать размер стека при его создании. Все члены класса Stack, кроме методов push() и pop(), должны быть объявлены как private. (Подсказка: в качестве заготовки можете воспользоваться классом Queue, изменив в нем лишь способ доступа к данным.)

4. Предположим, имеется следующий класс:

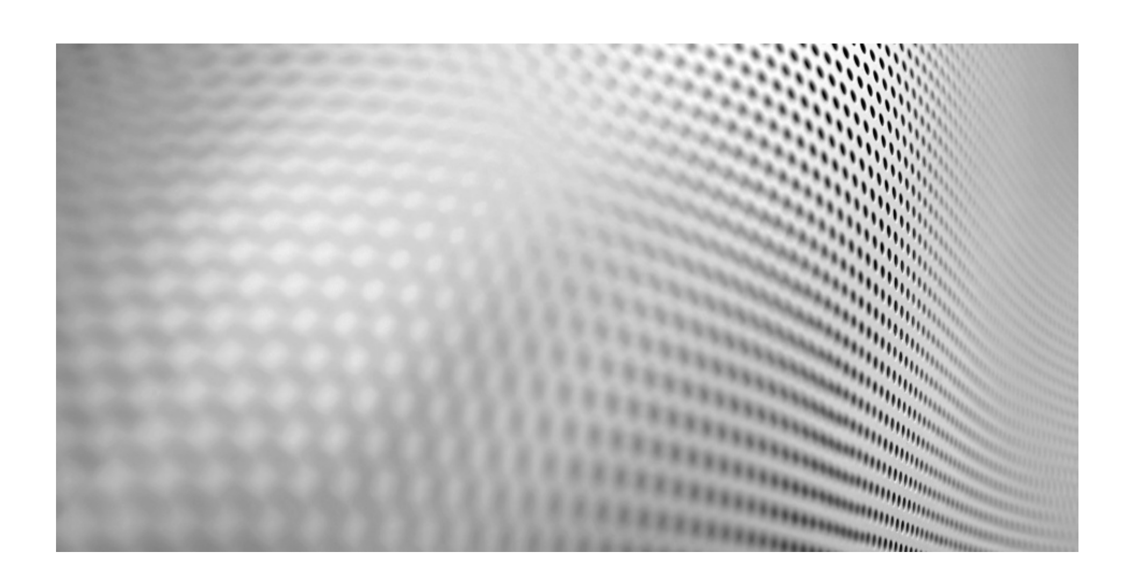
```
class Test {
   int a;
   Test(int i) { a = i; }
}
```

Напишите метод swap(), реализующий обмен содержимым между двумя объектами типа Test, на которые ссылаются две переменные данного типа.

5. Правильно ли написан следующий фрагмент кода?

```
class X {
   int meth(int a, int b) { ... }
   String meth(int a, int b) { ... }
```

6. Напишите рекурсивный метод, отображающий строку задом наперед.
7. Допустим, все объекты класса должны совместно использовать одну и ту же переменную. Как объявить такую переменную?
8. Для чего может понадобиться статический блок?
9. Что такое внутренний класс?
10. Допустим, требуется член класса, к которому могут обращаться только другие члены этого же класса. Какой модификатор доступа следует использовать в его объявлении?
11. Имя метода и список его параметров вместе составляют ______________ метода.
12. Если методу передается значение типа int, то в этом случае используется передача параметра по ____________.
13. Создайте метод sum(), имеющий список аргументов переменной длины и предназначенный для суммирования передаваемых ему значений типа int. Метод должен возвращать результат суммирования. Продемонстрируйте работу этого метода.
14. Можно ли перегружать методы с переменным числом аргументов?
15. Приведите пример вызова перегруженного метода с переменным числом аргументов, демонстрирующий возникновение неоднозначности.



Глава 7

Наследование

В этой главе...

- Основы наследования
- Способы вызова конструктора суперкласса
- Порядок обращения к членам суперкласса с помощью ключевого слова `super`
- Создание многоуровневой иерархии классов
- Вызов конструкторов
- Ссылки на объекты подкласса из переменной суперкласса
- Методика переопределения методов
- Использование переопределяемых методов для организации динамического доступа
- Абстрактные классы
- Использование ключевого слова `final`
- Класс `Object`

Одним из трех фундаментальных принципов объектно-ориентированного программирования является наследование, с помощью которого создаются иерархические классификации. На основе наследования можно создать общий класс, определяющий обобщенные характеристики для множества родственных элементов. Затем этот класс может наследоваться другими, более специализированными классами, каждый из которых будет добавлять собственные уникальные характеристики.

В соответствии с терминологией Java наследуемый класс называют *суперклассом*, а наследующий — *подклассом*. Таким образом, подкласс является специализированной версией суперкласса, которая наследует все переменные и методы суперкласса, дополняя их собственными, уникальными элементами.

Основы наследования

Чтобы наследовать класс в Java, достаточно включить его имя в объявление другого класса, используя ключевое слово `extends`. Таким образом, подкласс расширяет суперкласс, дополняя его собственными элементами.

Рассмотрим простой пример, наглядно демонстрирующий некоторые свойства наследования. Приведенная ниже программа создает суперкласс `TwoDShape`, в котором хранятся сведения о ширине и высоте двумерного

объекта, а также его подкласс `Triangle`. Обратите внимание на использование ключевого слова `extends` при создании подкласса.

```
// Простая иерархия классов

// Класс, описывающий двумерные объекты
class TwoDShape {
   double width;
   double height;

   void showDim() {
     System.out.println("Ширина и высота - " + width + " и " +
                        height);
   }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
   String style;

   double area() {
      return width * height / 2;
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}

class Shapes {
   public static void main(String args[]) {
      Triangle t1 = new Triangle();
      Triangle t2 = new Triangle();

      t1.width = 4.0;
      t1.height = 4.0;
      t1.style = "закрашенный";

      t2.width = 8.0;
      t2.height = 12.0;
      t2.style = "контурный";

      System.out.println("Информация о t1: ");
      t1.showStyle();
      t1.showDim();
      System.out.println("Площадь - " + t1.area());

      System.out.println();

      System.out.println("Информация о t2: ");
      t2.showStyle();
      t2.showDim();
```

Класс `Triangle` наследует класс `TwoDShape`

Из класса `Triangle` можно обращаться к членам класса `TwoDShape` так, как если бы это были его собственные члены

Объектам типа `Triangle` доступны все члены класса `Triangle`, даже те, которые унаследованы от класса `TwoDShape`

```
        System.out.println("Площадь - " + t2.area());
    }
}
```

Выполнение этой программы дает следующий результат.

```
Информация о t1:
Треугольник закрашенный
Ширина и высота - 4.0 и 4.0
Площадь - 8.0

Информация о t2:
Треугольник контурный
Ширина и высота - 8.0 и 12.0
Площадь - 48.0
```

В классе TwoDShape определены свойства обобщенной двумерной фигуры, частными случаями которой могут быть квадрат, треугольник, прямоугольник и т.п. Класс Triangle является конкретной разновидностью класса TwoDShape, в данном случае это треугольник. Класс Triangle включает в себя все элементы класса TwoDShape, а также поле style и методы area() и showStyle(). Описание стиля оформления треугольника хранится в переменной экземпляра style. В этой переменной может храниться любая строка, которая описывает треугольник, например "закрашенный", "контурный", "прозрачный", "равнобедренный" или "скругленный". Метод area() вычисляет и возвращает площадь треугольника, а метод showStyle() отображает стиль оформления треугольника.

Поскольку класс Triangle включает все члены суперкласса TwoDShape, в теле метода area() доступны переменные экземпляра width и height. Кроме того, объекты t1 и t2 в методе main() могут непосредственно обращаться к переменным width и height, как если бы они принадлежали классу Triangle. На рис. 7.1 схематически показано, каким образом суперкласс TwoDShape включается в состав класса Triangle.

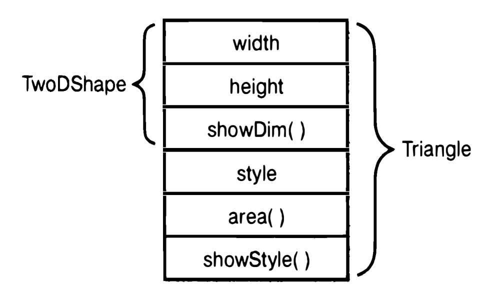


Рис. 7.1. Структура класса Triangle

Несмотря на то что TwoDShape является суперклассом для класса Triangle, он все равно остается независимым классом. Тот факт, что один класс является суперклассом другого, вовсе не исключает возможности его непосредственного использования. Например, следующий фрагмент кода является корректным.

```
TwoDShape shape = new TwoDShape();

shape.width = 10;
shape.height = 20;

shape.showDim();
```

Разумеется, объекту типа TwoDShape ничего не известно о подклассах своего класса TwoDShape, и он не может к ним обращаться.

Ниже приведена общая форма объявления класса, который наследует суперкласс.

```
class имя_подкласса extends имя_суперкласса {
   // тело класса
}
```

Для каждого создаваемого подкласса можно указать лишь один суперкласс. Множественное наследование в Java не поддерживается, т.е. у подкласса не может быть несколько суперклассов. (Этим Java отличается от языка C++, где допускается наследование одновременно нескольких классов. Не забывайте об этом, если вам когда-либо придется преобразовывать код C++ в код Java.) С другой стороны, в Java допускается многоуровневая иерархия, в которой один подкласс является суперклассом другого подкласса. И конечно же, класс не может быть суперклассом по отношению к самому себе.

Основное преимущество наследования заключается в следующем: создав суперкласс, в котором определены общие для множества объектов свойства, вы сможете использовать его для создания любого количества более специализированных подклассов. Каждый подкласс добавляет собственный набор специфических для него атрибутов в соответствии с той или иной задачей. В качестве примера ниже приведен еще один подкласс, который наследует суперкласс TwoDShape и инкапсулирует прямоугольники.

```
// Подкласс для представления прямоугольников,
// производный от класса TwoDShape
class Rectangle extends TwoDShape {
   boolean isSquare() {
      if(width == height) return true;
      return false;
   }

   double area() {
      return width * height;
   }
}
```

Класс Rectangle включает все члены класса TwoDShape. Кроме того, он содержит метод isSquare(), определяющий, является ли прямоугольник квадратом, а также метод area(), вычисляющий площадь прямоугольника.

Наследование и доступ к членам класса

Как отмечалось в главе 6, с целью исключения несанкционированного доступа к членам класса их часто объявляют как закрытые, используя для этого модификатор доступа `private`. Наследование класса *не отменяет* ограничений, налагаемых на доступ к закрытым членам класса. Поэтому, несмотря на то что в подкласс автоматически включаются все члены его суперкласса, доступ к закрытым членам суперкласса ему запрещен. Так, если переменные экземпляра `width` и `height` в классе `TwoDShape` объявить как закрытые (см. ниже), это предотвратит возможность доступа к ним из класса `Triangle`.

```
// Закрытые члены класса не наследуются

// Этот код не пройдет компиляцию

// Класс, описывающий двумерные объекты
class TwoDShape {
   private double width;  // теперь эти переменные
   private double height; // объявлены как закрытые

   void showDim() {
      System.out.println("Ширина и высота - " + width + " и " +
                         height);
   }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
   String style;

   double area() {
      return width * height / 2; // Ошибка: доступ запрещен!
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}
```

Доступ к членам суперкласса, объявленным как закрытые, невозможен

Класс `Triangle` не сможет быть скомпилирован, поскольку ссылки на переменные `width` и `height` в методе `area()` нарушают правила доступа. Эти переменные объявлены закрытыми (`private`), поэтому они доступны только членам собственного класса. Для подклассов доступ к ним невозможен.

Помните о том, что член класса, объявленный как закрытый, недоступен за пределами своего класса. Это ограничение распространяется на все подклассы данного класса.

На первый взгляд, ограничение на доступ к закрытым членам суперкласса из подкласса кажется неудобным и ведет к невозможности их использования.

Однако это вовсе не так. Как пояснялось в главе 6, для обращения к закрытым членам класса в программах на Java обычно используют специальные методы доступа. Ниже в качестве примера приведены видоизмененные классы `TwoDShape` и `Triangle`, в которых методы доступа применяются для обращения к переменным экземпляра `width` и `height`.

```
// Использование методов доступа для установки и
// получения значений закрытых членов.

// Класс, описывающий двумерные объекты
class TwoDShape {
   private double width; // теперь эти переменные
   private double height; // объявлены как закрытые

   // Методы доступа к закрытым переменным экземпляра width и height
   double getWidth() { return width; }
   double getHeight() { return height; }   ◄—— Методы доступа к переменным экземпляра width и height
   void setWidth(double w) { width = w; }
   void setHeight(double h) { height = h; }

   void showDim() {
      System.out.println("Ширина и высота - " + width + " и " +
                         height);
   }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
   String style;
                                  Использование методов доступа, предоставляемых суперклассом
   double area() {                ▼
      return getWidth() * getHeight() / 2;
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}

class Shapes2 {
   public static void main(String args[]) {
      Triangle t1 = new Triangle();
      Triangle t2 = new Triangle();

      t1.setWidth(4.0);
      t1.setHeight(4.0);
      t1.style = "закрашенный";

      t2.setWidth(8.0);
      t2.setHeight(12.0);
      t2.style = "контурный";
```

```
        System.out.println("Информация о t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Площадь - " + t1.area());

        System.out.println();

        System.out.println("Информация о t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Площадь - " + t2.area());
    }
}
```

Спросим у эксперта

ВОПРОС. В каких случаях переменную экземпляра нужно объявлять закрытой?

ОТВЕТ. Не существует четко сформулированных правил, позволяющих принять безошибочное решение по данному вопросу. Следует лишь придерживаться двух общих принципов. Во-первых, если переменная экземпляра используется только методами, определенными в классе, то она должна быть закрытой. И во-вторых, если значение переменной экземпляра не должно выходить за определенные границы, ее следует объявить как закрытую, а обращение к ней выполнять с помощью специальных методов доступа. Подобным образом можно предотвратить присваивание недопустимых значений переменной.

Конструкторы и наследование

В иерархии классов допускается, чтобы суперклассы и подклассы имели собственные конструкторы. В связи с этим возникает вопрос, какой именно конструктор отвечает за создание объекта подкласса: конструктор суперкласса, конструктор подкласса или же оба одновременно? На этот вопрос можно ответить так: конструктор суперкласса используется для построения родительской части объекта, а конструктор подкласса — для остальной его части. И в этом есть своя логика, поскольку суперклассу неизвестны и недоступны любые собственные члены подкласса, а значит, каждая из указанных частей объекта должна конструироваться по отдельности. В приведенных выше примерах этот вопрос не возникал, поскольку они базировались на автоматически создаваемых конструкторах, используемых по умолчанию. Но на практике в большинстве случаев конструкторы следует определять явным образом.

Если конструктор определен только в подклассе, то все происходит очень просто: конструируется объект подкласса, а родительская часть объекта автоматически создается конструктором суперкласса, используемым по умолчанию.

В качестве примера рассмотрим приведенный ниже переработанный вариант класса `Triangle`, в котором определяется собственный конструктор, вследствие чего члену `style` этого класса придается статус закрытого.

```
// Добавление конструктора в класс Triangle

// Класс, описывающий двумерные объекты
class TwoDShape {
   private double width;  // теперь эти переменные
   private double height; // объявлены как закрытые

   // Методы доступа к переменным экземпляра width и height
   double getWidth() { return width; }
   double getHeight() { return height; }
   void setWidth(double w) { width = w; }
   void setHeight(double h) { height = h; }

   void showDim() {
      System.out.println("Ширина и высота - " + width + " и " +
                         height);
   }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
   private String style;

   // Конструктор
   Triangle(String s, double w, double h) {
      setWidth(w);
      setHeight(h); ◄——— Инициализация части объекта,
                         соответствующей классу TwoDShape

      style = s;
   }

   double area() {
      return getWidth() * getHeight() / 2;
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}

class Shapes3 {
   public static void main(String args[]) {
      Triangle t1 = new Triangle("закрашенный", 4.0, 4.0);
      Triangle t2 = new Triangle("контурный", 8.0, 12.0);
```

```
        System.out.println("Информация о t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Площадь - " + t1.area());

        System.out.println();

        System.out.println("Информация о t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Площадь - " + t2.area());
    }
}
```

В данном случае конструктор класса Triangle инициализирует собственное поле style и унаследованные члены класса TwoDClass.

Если конструкторы объявлены как в подклассе, так и в суперклассе, то все немного усложняется, поскольку в этом случае будут выполняться оба конструктора. При этом на помощь приходит ключевое слово super, которое может применяться в двух общих формах. Первая форма используется для вызова конструктора суперкласса, а вторая — для доступа к членам суперкласса, скрытых членами подкласса. Рассмотрим первое из указанных применений ключевого слова super.

Использование ключевого слова super для вызова конструктора суперкласса

Для вызова конструктора суперкласса из подкласса используется следующий общий синтаксис ключевого слова super:

```
super(список_параметров);
```

где *список_параметров* определяет параметры, используемые конструктором суперкласса. Вызов конструктора super() всегда должен быть первой инструкцией в теле конструктора подкласса. Проиллюстрируем использование вызова super() на примере приведенной ниже программы, включающей видоизмененную версию класса TwoDShape, в которой определен конструктор, инициализирующий переменные экземпляра width и height.

```
// Добавление конструкторов в класс TwoDShape
class TwoDShape {
    private double width;
    private double height;

    // Параметризированный конструктор
    TwoDShape(double w, double h) { ◄——— Конструктор класса TwoDShape
        width = w;
        height = h;
    }
```

```
   // Методы доступа к переменным экземпляра width и height
   double getWidth() { return width; }
   double getHeight() { return height; }
   void setWidth(double w) { width = w; }
   void setHeight(double h) { height = h; }

   void showDim() {
      System.out.println("Ширина и высота - " + width + " и " +
                         height);
   }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
   private String style;

   Triangle(String s, double w, double h) {
      super(w, h); // вызов конструктора суперкласса

      style = s;
   }

   double area() {
      return getWidth() * getHeight() / 2;
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}

class Shapes4 {
   public static void main(String args[]) {
      Triangle t1 = new Triangle("закрашенный", 4.0, 4.0);
      Triangle t2 = new Triangle("контурный", 8.0, 12.0);

      System.out.println("Информация о t1: ");
      t1.showStyle();
      t1.showDim();
      System.out.println("Площадь - " + t1.area());

      System.out.println();

      System.out.println("Информация о t2: ");
      t2.showStyle();
      t2.showDim();
      System.out.println("Площадь - " + t2.area());
   }
}
```

Использование оператора `super()` **для вызова конструктора класса** `TwoDShape`

В конструкторе Triangle осуществляется вызов конструктора super() с параметрами w и h. Это приводит к тому, что управление получает конструктор TwoDShape(), инициализирующий переменные width и height данными значениями, благодаря чему класс Triangle не должен самостоятельно инициализировать элементы суперкласса. Ему остается инициализировать только собственную переменную экземпляра style. Конструктору TwoDShape() предоставляется возможность создать соответствующий объект так, как требуется для данного класса. Более того, в суперклассе TwoDShape можно реализовать функции, о которых подклассам ничего не будет известно. Благодаря этому повышается степень отказоустойчивости кода.

Вызов super() позволяет использовать любую форму конструктора, определенную в суперклассе. В данном случае выбирается тот вариант конструктора, который соответствует указанным аргументам. Ниже в качестве примера приведены расширенные версии классов TwoDShape и Triangle, которые содержат конструкторы, заданные по умолчанию, и конструкторы, имеющие один и более аргументов.

```
// Добавление дополнительных конструкторов в класс TwoDShape
class TwoDShape {
   private double width;
   private double height;

   // Конструктор, заданный по умолчанию
   TwoDShape() {
      width = height = 0.0;
   }

   // Параметризированный конструктор
   TwoDShape(double w, double h) {
      width = w;
      height = h;
   }

   // Создание объекта с одинаковыми значениями
   // переменных экземпляра width и height
   TwoDShape(double x) {
      width = height = x;
   }

   // Методы доступа к переменным экземпляра width и height
   double getWidth() { return width; }
   double getHeight() { return height; }
   void setWidth(double w) { width = w; }
   void setHeight(double h) { height = h; }

   void showDim() {
      System.out.println("Ширина и высота - " + width + " и " +
                         height);
   }
}
```

```
// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
   private String style;

   // Конструктор по умолчанию
   Triangle() {
      super(); // вызов конструктора суперкласса по умолчанию
      style = "none";
   }

   // Конструктор
   Triangle(String s, double w, double h) {
      super(w, h); // вызов конструктора суперкласса с
                   // двумя аргументами

      style = s;
   }

   // Конструктор с одним аргументом
   Triangle(double x) {
      super(x); // вызов конструктора суперкласса
                // с одним аргументом

      style = "закрашенный";
   }

   double area() {
      return getWidth() * getHeight() / 2;
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}

class Shapes5 {
   public static void main(String args[]) {
      Triangle t1 = new Triangle();
      Triangle t2 = new Triangle("контурный", 8.0, 12.0);
      Triangle t3 = new Triangle(4.0);

      t1 = t2;

      System.out.println("Информация о t1: ");
      t1.showStyle();
      t1.showDim();
      System.out.println("Площадь - " + t1.area());

      System.out.println();

      System.out.println("Информация о t2: ");
      t2.showStyle();
```

Использование метода super() для вызова разных форм конструктора TwoDShape()

```
        t2.showDim();
        System.out.println("Площадь - " + t2.area());

        System.out.println();

        System.out.println("Информация о t3: ");
        t3.showStyle();
        t3.showDim();
        System.out.println("Площадь - " + t3.area());

        System.out.println();
    }
}
```

Выполнение этого варианта программы приведет к следующему результату.

```
Информация о t1:
Треугольник контурный
Ширина и высота - 8.0 и 12.0
Площадь - 48.0

Информация о t2:
Треугольник контурный
Ширина и высота - 8.0 и 12.0
Площадь - 48.0

Информация о t3:
Треугольник закрашенный
Ширина и высота - 4.0 и 4.0
Площадь - 8.0
```

Еще раз напомним основные особенности вызова конструктора `super()`. Если этот вызов присутствует в конструкторе подкласса, то происходит обращение к конструктору его непосредственного суперкласса. Таким образом, вызывается конструктор того класса, который непосредственно породил вызывающий класс. Это справедливо и при многоуровневой иерархии. Кроме того, вызов конструктора `super()` должен быть первой инструкцией в теле конструктора подкласса.

Использование ключевого слова super для доступа к членам суперкласса

Существует еще одна общая форма ключевого слова `super`, которая применяется подобно ключевому слову `this`, но ссылается на суперкласс данного класса. Эта общая форма обращения к члену суперкласса имеет следующий вид:

```
super.член_класса
```

где *член_класса* обозначает метод или переменную экземпляра.

Данная форма ключевого слова `super` используется в тех случаях, когда член подкласса скрывает член суперкласса. Рассмотрим следующий пример простой иерархии классов.

```
// Использование ключевого слова super
// для предотвращения сокрытия имен
class A {
   int i;
}

// Создание подкласса, расширяющего класс A
class B extends A {
   int i; // эта переменная i скрывает переменную i из класса A

   B(int a, int b) {
      super.i = a; // переменная i из класса A  ◄──── Здесь super.i ссылается
      i = b;       // переменная i из класса B        на переменную i из класса A
   }

   void show() {
      System.out.println("i в суперклассе: " + super.i);
      System.out.println("i в подклассе: " + i);
   }
}

class UseSuper {
   public static void main(String args[]) {
      B subOb = new B(1, 2);

      subOb.show();
   }
}
```

Выполнение этой программы даст следующий результат.

```
i в суперклассе: 1
i в подклассе: 2
```

Несмотря на то что переменная экземпляра `i` в классе `B` скрывает одноименную переменную в классе `A`, ключевое слово `super` позволяет обращаться к переменной `i` из суперкласса. Аналогичным образом ключевое слово `super` можно использовать для вызова методов суперкласса, скрываемых методами подкласса.

Упражнение 7.1 Расширение класса Vehicle

`TruckDemo.javajava`

Для того чтобы продемонстрировать возможности наследования, расширим класс `Vehicle`, созданный в главе 4. Напомним, что класс `Vehicle` инкапсулирует данные о транспортных средствах и, в частности, сведения о количестве пассажиров, объеме топливного бака и расходе топлива. Воспользуемся классом `Vehicle` в качестве заготовки, на основе которой будут созданы более специализированные классы. Например, к категории транспортных средств, помимо всех прочих, относятся грузовики. Одной из важных характеристик грузовика является его грузоподъемность.

Поэтому для создания класса `Truck` можно расширить класс `Vehicle`, добавив переменную экземпляра, хранящую сведения о допустимом весе перевозимого груза. В этом упражнении переменные экземпляра объявляются в классе `Vehicle` как закрытые (`private`), а для обращения к ним используются специальные методы доступа. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте новый файл `TruckDemo.java` и скопируйте в него исходный код последней версии класса `Vehicle`, разработанной в главе 4.
2. Создайте класс `Truck`, исходный код которого приведен ниже.

```
// Расширение класса Vehicle для грузовиков
class Truck extends Vehicle {
   private int cargocap; // грузоподъемность, выраженная в фунтах

   // Конструктор класса Truck
   Truck(int p, int f, int m, int c) {
      // Инициализация членов класса Vehicle
      // с использованием конструктора этого класса
      super(p, f, m);
      cargocap = c;
   }

   // Методы доступа к переменной cargocap
   int getCargo() { return cargocap; }
   void putCargo(int c) { cargocap = c; }
}
```

   В данном случае класс `Truck` наследует класс `Vehicle`. В класс `Truck` добавлены новые члены `cargocap`, `getCargo()` и `putCargo()`. Кроме того, он содержит все элементы, определенные в классе `Vehicle`.
3. Объявим закрытые переменные экземпляра в классе `Vehicle`.

```
private int passengers; // количество пассажиров
private int fuelcap;    // объем топливного бака (в галлонах)
private int mpg;        // потребление топлива (в милях на галлон)
```

4. Ниже приведен полный исходный код программы, в которой демонстрируется использование класса `Truck`.

```
// Упражнение 7.1
//
// Создание подкласса класса Vehicle для грузовиков
class Vehicle {
   private int passengers; // количество пассажиров
   private int fuelcap;    // объем топливного бака (в галлонах)
   private int mpg;        // потребление топлива (в милях на галлон)

   // Конструктор класса Vehicle
   Vehicle(int p, int f, int m) {
      passengers = p;
```

```
      fuelcap = f;
      mpg = m;
   }

   // Дальность поездки транспортного средства
   int range() {
      return mpg * fuelcap;
   }

   // Вычисление объема топлива, требуемого
   // для прохождения заданного пути
   double fuelneeded(int miles) {
      return (double) miles / mpg;
   }

   // Методы доступа к переменным экземпляра
   int getPassengers() { return passengers; }
   void setPassengers(int p) { passengers = p; }
   int getFuelcap() { return fuelcap; }
   void setFuelcap(int f) { fuelcap = f; }
   int getMpg() { return mpg; }
   void setMpg(int m) { mpg = m; }
}

// Расширение класса Vehicle для грузовиков
class Truck extends Vehicle {
   private int cargocap; // грузоподъемность, выраженная в фунтах

   // Конструктор класса Truck
   Truck(int p, int f, int m, int c) {
      // Инициализация членов класса Vehicle
      // с использованием конструктора этого класса
      super(p, f, m);

      cargocap = c;
   }

   // Методы доступа к переменной cargocap
   int getCargo() { return cargocap; }
   void putCargo(int c) { cargocap = c; }
}

class TruckDemo {
   public static void main(String args[]) {
      // Создание ряда новых объектов типа Truck
      Truck semi = new Truck(2, 200, 7, 44000);
      Truck pickup = new Truck(3, 28, 15, 2000);
      double gallons;
      int dist = 252;

      gallons = semi.fuelneeded(dist);
```

```
        System.out.println("Грузовик может перевезти " +
                           semi.getCargo() + " фунтов.");
        System.out.println("Для преодоления " + dist +
                           " миль грузовику требуется " + gallons +
                           " галлонов топлива.\n");

        gallons = pickup.fuelneeded(dist);

        System.out.println("Пикап может перевезти " +
                           pickup.getCargo() + " фунтов.");
        System.out.println("Для преодоления " + dist +
                           " миль пикапу требуется " + gallons +
                           " галлонов топлива.");
    }
}
```

5. Ниже приведен результат выполнения данной программы.

```
Грузовик может перевезти 44000 фунтов.
Для преодоления 252 миль грузовику требуется 36.0 галлонов топлива.

Пикап может перевезти 2000 фунтов.
Для преодоления 252 миль пикапу требуется 16.8 галлонов топлива.
```

6. На основе класса `Vehicle` можно создать немало других подклассов. Например, в приведенной ниже заготовке класса, описывающего внедорожники, предусмотрена переменная, содержащая величину дорожного просвета для автомобиля.

```
// Создание класса, описывающего внедорожники
class OffRoad extends Vehicle {
    private int groundClearance; // дорожный просвет, выраженный
                                 // в дюймах

    // ...
}
```

На основе суперкласса, определяющего общие свойства некоторых объектов, можно создать специализированные подклассы. Каждый подкласс дополняет свойства суперкласса собственными уникальными свойствами. В этом и состоит сущность наследования.

Создание многоуровневой иерархии классов

До сих пор применялись простые иерархии классов, которые состояли только из суперкласса и подкласса. Однако Java позволяет создавать иерархии, состоящие из произвольного количества уровней наследования. Как уже упоминалось выше, многоуровневая иерархия идеально подходит для использования одного подкласса в качестве суперкласса для другого подкласса. Так, если имеются три класса, `A`, `B` и `C`, то класс `C` может наследовать все характеристики

класса B, а тот, в свою очередь, все характеристики класса A. В подобных случаях каждый подкласс наследует характерные особенности всех своих суперклассов. В частности, класс C наследует все члены классов B и A.

Для того чтобы стало понятнее назначение многоуровневой иерархии, рассмотрим следующий пример программы, в которой подкласс Triangle выступает в качестве суперкласса для класса ColorTriangle. Класс ColorTriangle наследует все свойства классов Triangle и TwoDShape, а также включает поле color, задающее цвет треугольника.

```
// Многоуровневая иерархия
class TwoDShape {
   private double width;
   private double height;

   // Конструктор по умолчанию
   TwoDShape() {
      width = height = 0.0;
   }

   // Параметризированный конструктор
   TwoDShape(double w, double h) {
      width = w;
      height = h;
   }

   // Создание объекта с одинаковыми значениями
   // переменных экземпляра width и height
   TwoDShape(double x) {
      width = height = x;
   }

   // Методы доступа к переменным экземпляра width и height
   double getWidth() { return width; }
   double getHeight() { return height; }
   void setWidth(double w) { width = w; }
   void setHeight(double h) { height = h; }

   void showDim() {
      System.out.println("Ширина и высота - " + width + " и " +
                         height);
   }
}

// Расширение класса TwoDShape
class Triangle extends TwoDShape {
   private String style;

   // Конструктор по умолчанию
   Triangle() {
      super();
```

```
      style = "none";
   }

   Triangle(String s, double w, double h) {
      super(w, h); // вызов конструктора суперкласса

      style = s;
   }

   // Конструктор с одним аргументом для построения треугольника
   Triangle(double x) {
      super(x); // вызов конструктора суперкласса

      style = "закрашенный";
   }

   double area() {
      return getWidth() * getHeight() / 2;
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}

// Расширение класса Triangle
class ColorTriangle extends Triangle {
   private String color;

   ColorTriangle(String c, String s, double w, double h) {
      super(s, w, h);

      color = c;
   }

   String getColor() { return color; }

   void showColor() {
      System.out.println("Цвет - " + color);
   }
}

class Shapes6 {
   public static void main(String args[]) {
      ColorTriangle t1 =
         new ColorTriangle("Синий", "контурный", 8.0, 12.0);
      ColorTriangle t2 =
         new ColorTriangle("Красный", "закрашенный", 2.0, 2.0);

      System.out.println("Информация о t1: ");
      t1.showStyle();
```

Класс ColorTriangle наследует класс Triangle, производный от класса TwoDShape, и поэтому включает все члены классов Triangle и TwoDShape

```
        t1.showDim();
        t1.showColor();
        System.out.println("Площадь - " + t1.area());

        System.out.println();

        System.out.println("Информация о t2: ");
        t2.showStyle();
        t2.showDim(); ◄──── Объект типа ColorTriangle может вызывать как собственные методы, так и методы суперклассов
        t2.showColor();
        System.out.println("Площадь - " + t2.area());
    }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
Информация о t1:
Треугольник контурный
Ширина и высота - 8.0 и 12.0
Цвет - Синий
Площадь - 48.0

Информация о t2:
Треугольник закрашенный
Ширина и высота - 2.0 и 2.0
Цвет - Красный
Площадь - 2.0
```

Благодаря наследованию в классе ColorTriangle можно использовать ранее определенные классы Triangle и TwoDShape, дополняя их лишь полями, необходимыми для конкретного применения класса ColorTriangle. Таким образом, наследование способствует повторному использованию кода.

Данный пример демонстрирует еще одну важную деталь: вызов метода super() всегда означает обращение к конструктору ближайшего суперкласса. Иными словами, вызов super() в классе ColorTriangle означает вызов конструктора класса Triangle, а в классе Triangle — вызов конструктора класса TwoDShape. Если в иерархии классов для конструктора суперкласса предусмотрены параметры, то все подклассы должны передавать их вверх по иерархической структуре. Это правило действует независимо от того, нужны ли параметры самому подклассу.

Очередность вызова конструкторов

После прочтения материала, посвященного наследованию и иерархии классов, у читателей может возникнуть вопрос, когда именно создается объект подкласса и какой именно конструктор выполняется первым: тот, который определен в подклассе, или тот, который определен в суперклассе? Например, если имеется суперкласс A и подкласс B, то что будет вызываться раньше: конструктор класса A или конструктор класса B? Ответ на этот вопрос заключается в том,

что в иерархии классов конструкторы вызываются в порядке наследования, начиная с суперкласса и заканчивая подклассом. Более того, метод super() должен быть первой инструкцией в конструкторе подкласса, и поэтому порядок, в котором вызываются конструкторы, остается неизменным, независимо от того, используется ли вызов метода super() или нет. Если вызов метода super() отсутствует, то выполняется конструктор каждого суперкласса по умолчанию (т.е. конструктор без параметров). Порядок вызова конструкторов демонстрируется на примере следующей программы.

```
// Демонстрация очередности вызова конструкторов

// Создание суперкласса
class A {
   A() {
      System.out.println("Конструктор A");
   }
}

// Создание подкласса в результате расширения класса A
class B extends A {
   B() {
      System.out.println("Конструктор B");
   }
}

// Создание подкласса в результате расширения класса B
class C extends B {
   C() {
      System.out.println("Конструктор C");
   }
}

class OrderOfConstruction {
   public static void main(String args[]) {
      C c = new C();
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
Конструктор A
Конструктор B
Конструктор C
```

Как видите, конструкторы вызываются в порядке наследования классов.

Если вникнуть в суть вопроса, то можно прийти к выводу, что вызов конструкторов в порядке наследования классов имеет определенный смысл. Ведь суперклассу ничего не известно ни об одном из производных от него подклассов, и поэтому любая инициализация, которая требуется его членам, не только должна осуществляться независимо от инициализации членов подкласса, но и, возможно, является необходимой подготовительной операцией, требуемой для выполнения этого процесса. Следовательно, она будет выполняться первой.

Ссылки на суперкласс и объекты подклассов

Ранее уже упоминалось о том, что Java является строго типизированным языком программирования. Помимо стандартных преобразований и автоматического повышения простых типов данных, в этом языке четко соблюдается принцип совместимости типов. Это означает, что переменная, ссылающаяся на объект класса одного типа, как правило, не может ссылаться на объект класса другого типа. В качестве примера рассмотрим следующую простую программу.

```
// Этот код не пройдет компиляцию
class X {
   int a;

   X(int i) { a = i; }
}

class Y {
   int a;

   Y(int i) { a = i; }
}

class IncompatibleRef {
   public static void main(String args[]) {
      X x = new X(10);
      X x2;
      Y y = new Y(5);

      x2 = x; // допустимо, поскольку обе переменные одного типа

      x2 = y; // ошибка, поскольку переменные разных типов
   }
}
```

Несмотря на то что классы X и Y содержат одинаковые члены, переменной типа X невозможно присвоить ссылку на объект типа Y, поскольку типы объектов отличаются. Вообще говоря, ссылочная переменная может указывать только на объекты своего типа.

Стоит отметить, что существует одно важное исключение из этого правила: ссылочной переменной суперкласса может быть присвоена ссылка на объект любого подкласса, производного от данного суперкласса. Таким образом, ссылку на объект суперкласса можно использовать для обращения к объектам соответствующих подклассов. Ниже приведен соответствующий пример.

```
// Обращение к объекту подкласса по ссылочной
// переменной суперкласса
class X {
   int a;

   X(int i) { a = i; }
}
```

```
class Y extends X {
   int b;

   Y(int i, int j) {
      super(j);
      b = i;
   }
}

class SupSubRef {
   public static void main(String args[]) {
      X x = new X(10);
      X x2;
      Y y = new Y(5, 6);

      x2 = x; // допустимо, поскольку обе переменные одного типа
      System.out.println("x2.a: " + x2.a);

      x2 = y; // по-прежнему допустимо по указанной выше причине
      System.out.println("x2.a: " + x2.a);

      // В классе X известны только члены класса X
      x2.a = 19; // допустимо
      // x2.b = 27; // ошибка, так как переменная b не
                    // является членом класса X
   }
}
```

Класс Y является подклассом X, поэтому переменные x2 и y могут ссылаться на один и тот же объект производного класса

В этом примере класс Y является подклассом X. Следовательно, переменной x2 можно присвоить ссылку на объект типа Y.

Следует особо подчеркнуть, что доступ к конкретным членам класса определяется типом ссылочной переменной, а не типом объекта, на который она ссылается. Это означает, что если ссылка на объект подкласса присваивается ссылочной переменной суперкласса, то последняя может быть использована для доступа только к тем частям данного объекта, которые определяются суперклассом. Именно поэтому переменной x2 недоступен член b класса Y, когда она ссылается на объект этого класса. И в этом есть своя логика, поскольку суперклассу ничего не известно о тех элементах, которые добавлены в его подкласс. Именно поэтому последняя строка кода в приведенном выше примере была закомментирована.

Несмотря на то что приведенные выше рассуждения могут показаться несколько отвлеченными, им соответствует ряд важных практических применений. Одно из них будет упомянуто ниже, а другое будет рассмотрено далее, когда речь пойдет о переопределении методов.

Один из самых важных моментов для присваивания ссылок на объекты подкласса переменным с типом суперкласса наступает тогда, когда конструкторы вызываются в иерархии классов. Как вам уже должно быть известно, в классе зачастую определяется конструктор, получающий объект собственного класса

в качестве параметра. Благодаря этому в классе может быть создана копия объекта. Этой особенностью можно воспользоваться в подклассах, производных от такого класса. В качестве примера рассмотрим описанные ниже версии классов `TwoDShape` и `Triangle`. В оба класса добавлены конструкторы, получающие объект своего класса в качестве параметра.

```
class TwoDShape {
   private double width;
   private double height;

   // Конструктор по умолчанию
   TwoDShape() {
      width = height = 0.0;
   }

   // Параметризированный конструктор
   TwoDShape(double w, double h) {
      width = w;
      height = h;
   }

   // Создание объекта с одинаковыми значениями
   // переменных экземпляра width и height
   TwoDShape(double x) {
      width = height = x;
   }

   // Создание одного объекта на основе другого
   TwoDShape(TwoDShape ob) {  ◄——— Конструирование объекта на основе другого объекта
      width = ob.width;
      height = ob.height;
   }

   // Методы доступа к переменным экземпляра width и height
   double getWidth() { return width; }
   double getHeight() { return height; }
   void setWidth(double w) { width = w; }
   void setHeight(double h) { height = h; }

   void showDim() {
      System.out.println("Ширина и высота - " + width + " и " +
                         height);
   }
}

// Подкласс, применяемый для представления треугольников
// и производный от класса TwoDShape
class Triangle extends TwoDShape {
   private String style;

   // Конструктор по умолчанию
   Triangle() {
```

```
      super();
      style = "none";
   }

   // Конструктор класса Triangle
   Triangle(String s, double w, double h) {
      super(w, h); // вызов конструктора суперкласса

      style = s;
   }

   // Конструктор с одним аргументом для построения треугольника
   Triangle(double x) {
      super(x); // вызов конструктора суперкласса

      style = "закрашенный";
   }

   // Создание одного объекта на основе другого
   Triangle(Triangle ob) {
      super(ob); // передача объекта конструктору класса TwoDShape
      style = ob.style;
   }

   double area() {
      return getWidth() * getHeight() / 2;
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}

class Shapes7 {
   public static void main(String args[]) {
      Triangle t1 = new Triangle("контурный", 8.0, 12.0);

      // создать копию объекта t1
      Triangle t2 = new Triangle(t1);

      System.out.println("Информация о t1: ");
      t1.showStyle();
      t1.showDim();
      System.out.println("Площадь - " + t1.area());

      System.out.println();

      System.out.println("Информация о t2: ");
      t2.showStyle();
      t2.showDim();
      System.out.println("Площадь - " + t2.area());
   }
}
```

Передача ссылки Triangle конструктору TwoDShape

В приведенном выше примере программы объект t2 создается на основе объекта t1, и потому эти два объекта идентичны. Результат выполнения данной программы выглядит следующим образом.

```
Информация о t1:
Треугольник контурный
Ширина и высота - 8.0 и 12.0
Площадь - 48.0

Информация о t2:
Треугольник контурный
Ширина и высота - 8.0 и 12.0
Площадь - 48.0
```

Обратите внимание на конструктор класса Triangle, код которого приведен ниже.

```
// Создание одного объекта на основе другого
Triangle(Triangle ob) {
   // Передача объекта конструктору класса TwoDShape
   super(ob);
   style = ob.style;
}
```

В качестве параметра данному конструктору передается объект Triangle, который затем с помощью вызова метода super() передается конструктору TwoDShape.

```
// Создание одного объекта на основе другого
TwoDShape(TwoDShape ob) {
   width = ob.width;
   height = ob.height;
}
```

Следует отметить, что конструктор TwoDshape() должен получить объект типа TwoDShape, но конструктор Triangle() передает ему объект типа Triangle. Несмотря на это, каких-либо проблем не возникает. Ведь, как упоминалось ранее, ссылочная переменная суперкласса может ссылаться на объект подкласса. Следовательно, конструктору TwoDShape() можно передать ссылку на экземпляр подкласса, производного от класса TwoDShape. Конструктор TwoDShape() инициализирует лишь те части передаваемого ему объекта подкласса, которые являются членами класса TwoDShape, и потому не имеет значения, содержит ли этот объект дополнительные члены, добавленные в производных подклассах.

Переопределение методов

В иерархии классов часто есть методы с одинаковой сигнатурой и одинаковым возвращаемым значением, как в суперклассе, так и в подклассе. В этом случае говорят, что метод суперкласса *переопределяется* в подклассе. Если

переопределяемый метод вызывается из подкласса, то он всегда будет ссылаться на версию метода, определенную в подклассе. Версия метода, определенная в суперклассе, скрывается. Рассмотрим в качестве примера следующую программу.

```
// Переопределение метода
class A {
   int i, j;

   A(int a, int b) {
      i = a;
      j = b;
   }

   // Отображение переменных i и j
   void show() {
      System.out.println("i и j: " + i + " " + j);
   }
}

class B extends A {
   int k;

   B(int a, int b, int c) {
      super(a, b);
      k = c;
   }

   // Отображение переменной k - переопределение метода show() в A
   void show() {  // Метод show() в B переопределяет метод show() в A
      System.out.println("k: " + k);
   }
}

class Override {
   public static void main(String args[]) {
      B subOb = new B(1, 2, 3);

      subOb.show(); // вызов метода show() из класса B
   }
}
```

Выполнение этой программы даст следующий результат:

```
k: 3
```

Если метод `show()` вызывается для объекта типа `B`, выбирается версия этого метода, определенная в классе `B`. Таким образом, версия метода `show()` в классе `B` переопределяет версию одноименного метода, объявленную в классе `A`.

Чтобы обратиться к исходной версии переопределяемого метода, т.е. к той, которая определена в суперклассе, следует воспользоваться ключевым словом `super`. Например, в приведенном ниже варианте класса `B` из метода `show()`

вызывается версия того же метода, определенная в суперклассе. При этом отображаются все переменные экземпляра.

```
class B extends A {
   int k;

   B(int a, int b, int c) {
      super(a, b);
      k = c;
   }

   void show() {
      super.show();
      System.out.println("k: " + k);
   }
}
```

С помощью ключевого слова super вызывается версия метода show(), определенная в суперклассе A

Если подставить новую версию метода show() в предыдущий вариант программы, результат ее выполнения изменится, и будет иметь следующий вид:

```
i и j: 1 2
k: 3
```

В данном случае super.show() — это вызов метода show(), определенного в суперклассе.

Переопределение метода имеет место только в том случае, когда сигнатуры переопределяемого и переопределяющего методов совпадают. В противном случае происходит обычная перегрузка методов. Рассмотрим следующую видоизмененную версию предыдущего примера.

```
/* Методы с разными сигнатурами не переопределяются,
   а перегружаются. */
class A {
   int i, j;

   A(int a, int b) {
      i = a;
      j = b;
   }

   // Отображение переменных i и j
   void show() {
      System.out.println("i и j: " + i + " " + j);
   }
}

// Создание подкласса путем расширения класса A
class B extends A {
   int k;

   B(int a, int b, int c) {
      super(a, b);
```

Поскольку сигнатуры разные, эта версия метода show() просто перегружает метод show() в суперклассе A

```
      k = c;
   }

   // Перегрузка метода show()
   void show(String msg) {
      System.out.println(msg + k);
   }
}

class Overload {
   public static void main(String args[]) {
      B subOb = new B(1, 2, 3);

      subOb.show("k: "); // вызов метода show() из класса B
      subOb.show(); // вызов метода show() из класса A
   }
}
```

Выполнение этой программы приведет к следующему результату.

```
k: 3
i и j: 1 2
```

На этот раз в версии метода `show()` из класса `B` предусмотрен строковый параметр. Из-за этого сигнатура данного метода отличается от сигнатуры метода `show()` из класса `A`, для которого параметры не предусмотрены. Соответственно, никакого переопределения метода не происходит.

Поддержка полиморфизма в переопределяемых методах

Несмотря на то что приведенные выше примеры позволили продемонстрировать специфику использования переопределенных методов, этого недостаточно для того, чтобы можно было в полной мере оценить, какие широкие возможности обеспечиваются данным механизмом. Действительно, если бы переопределение методов представляло собой не более чем некий свод соглашений относительно использования пространств имен, то все, о чем говорилось выше, можно было бы считать хотя и заслуживающей интереса, но малополезной с точки зрения практики особенностью языка программирования. Однако это далеко не так. Механизм переопределения методов лежит в основе одного из наиболее эффективных языковых средств Java — *динамической диспетчеризации методов*, обеспечивающей возможность поиска подходящей версии переопределенного метода во время выполнения программы (а не во время ее компиляции).

Вспомним очень важный принцип: ссылочная переменная суперкласса может ссылаться на объект подкласса. В Java этот принцип используется для вызова переопределяемых методов во время выполнения. Если вызов переопределенного метода осуществляется с использованием ссылки на суперкласс, то

исполняющая среда Java выбирает нужную версию метода на основании типа объекта, на который эта ссылка указывает в момент вызова. Ссылкам на различные типы объектов будут соответствовать вызовы различных версий переопределенного метода. Иными словами, *на этапе выполнения программы версия переопределенного метода выбирается в зависимости от типа объекта ссылки* (а не типа ссылочной переменной). Следовательно, если суперкласс содержит метод, переопределенный в подклассе, то будет вызываться метод, соответствующий тому объекту, на который указывает ссылочная переменная суперкласса.

Ниже приведен простой пример, демонстрирующий использование динамической диспетчеризации вызовов методов.

```
// Демонстрация динамической диспетчеризации методов
class Sup {
   void who() {
      System.out.println("who() в Sup");
   }
}

class Sub1 extends Sup {
   void who() {
      System.out.println("who() в Sub1");
   }
}

class Sub2 extends Sup {
   void who() {
      System.out.println("who() в Sub2");
   }
}

class DynDispDemo {
   public static void main(String args[]) {
      Sup superOb = new Sup();
      Sub1 subOb1 = new Sub1();
      Sub2 subOb2 = new Sub2();

      Sup supRef;

      supRef = superOb;
      supRef.who();   ◄──── В каждом из
                            этих вызовов
      supRef = subOb1;      выбор версии
      supRef.who();   ◄──── метода who()
                            осуществляется
      supRef = subOb2;      по типу объекта,
      supRef.who();   ◄──── на который
                            указывает ссылка
                            во время выполнения
   }
}
```

Результат выполнения этой программы выглядит следующим образом.

```
who() в Sup
who() в Sub1
who() в Sub2
```

В данном примере программы определяются суперкласс `Sup` и два его подкласса: `Sub1` и `Sub2`. В классе `Sup` объявляется метод `who()`, переопределяемый в подклассах, а в методе `main()` создаются объекты типа `Sup`, `Sub1` и `Sub2`. Там же объявляется переменная `supRef`, ссылающаяся на объект типа `Sup`. Затем переменной `supRef` в методе `main()` поочередно присваиваются ссылки на объекты разного типа, и далее эти ссылки используются для вызова метода `who()`. Как следует из результата выполнения данной программы, вызываемая версия метода `who()` определяется типом объекта, на который указывает переменная `supRef` в момент вызова, а не типом самой переменной.

Спросим у эксперта

ВОПРОС. Переопределяемые методы напоминают виртуальные функции в C++. Есть ли у них сходство?

ОТВЕТ. Да, есть. Читатели, знакомые с языком C++, заметят, что переопределяемые методы в Java применяются с той же целью и тем же способом, что и виртуальные функции в C++.

Для чего нужны переопределяемые методы

Как упоминалось выше, переопределяемые методы обеспечивают поддержку полиморфизма времени выполнения. Большое значение полиморфизма в объектно-ориентированных программах обусловлено тем, что благодаря ему можно объявлять в суперклассе методы, общие для всех его подклассов, а в самих подклассах определять конкретные реализации всех этих методов или некоторых из них. Переопределение методов — один из способов, которыми в Java реализуется принцип полиморфизма "один интерфейс — множество методов".

Залогом успешного применения полиморфизма является, в частности, понимание того, что суперклассы и подклассы образуют иерархию по степени увеличения специализации. При продуманной организации суперкласса он предоставляет своему подклассу все элементы, которыми тот может пользоваться непосредственно. В нем также определяются те методы, которые должны быть по-своему реализованы в производных классах. Таким образом, подклассы получают достаточную свободу в определении собственных методов, реализуя в то же время согласованный интерфейс. Сочетая наследование с переопределением методов, в суперклассе можно определить общую форму методов для использования во всех его подклассах.

Демонстрация механизма переопределения методов на примере класса TwoDShape

Для того чтобы стало понятнее, насколько эффективным является механизм переопределения методов, продемонстрируем его применение на примере класса `TwoDShape`. В приведенных ранее примерах в каждом классе, наследующем класс `TwoDShape`, определялся метод `area()`. Теперь мы знаем, что в этом случае имеет смысл включить метод `area()` в состав класса `TwoDShape`, позволить каждому его подклассу переопределить этот метод и, в частности, реализовать вычисление площади в зависимости от конкретного типа геометрической фигуры. Именно такой подход и реализован в приведенном ниже примере программы. Для удобства в класс `TwoDShape` добавлено поле `name`. (Это упрощает написание демонстрационной программы.)

```
// Использование динамической диспетчеризации методов
class TwoDShape {
   private double width;
   private double height;
   private String name;

   // Конструктор по умолчанию
   TwoDShape() {
      width = height = 0.0;
      name = "none";
   }

   // Параметризированный конструктор
   TwoDShape(double w, double h, String n) {
      width = w;
      height = h;
      name = n;
   }

   // Создание объекта с одинаковыми значениями
   // переменных экземпляра width и height
   TwoDShape(double x, String n) {
      width = height = x;
      name = n;
   }

   // Создание одного объекта на основе другого
   TwoDShape(TwoDShape ob) {
      width = ob.width;
      height = ob.height;
      name = ob.name;
   }

   // Методы доступа к переменным экземпляра width и height
   double getWidth() { return width; }
   double getHeight() { return height; }
```

```
   void setWidth(double w) { width = w; }
   void setHeight(double h) { height = h; }

   String getName() { return name; }

   void showDim() {
      System.out.println("Ширина и высота - " + width + " и " +
                         height);
   }
                              Метод area(), определенный классом TwoDShape
   double area() { ◄──────────────┘
      System.out.println("Метод area() должен быть переопределен");
      return 0.0;
   }
}

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
   private String style;

   // Конструктор по умолчанию
   Triangle() {
      super();
      style = "none";
   }

   // Конструктор класса Triangle
   Triangle(String s, double w, double h) {
      super(w, h, "треугольник");
      style = s;
   }

   // Конструктор с одним аргументом для построения треугольника
   Triangle(double x) {
      super(x, "треугольник"); // вызов конструктора суперкласса
      style = "закрашенный";
   }

   // Создание одного объекта на основе другого
   Triangle(Triangle ob) {
      super(ob); // передача объекта конструктору класса TwoDShape
      style = ob.style;
   }

   // Переопределение метода area() для класса Triangle
   double area() { ◄──────────────────── Переопределение метода area()
      return getWidth() * getHeight() / 2;  для класса Triangle
   }

   void showStyle() {
      System.out.println("Треугольник " + style);
   }
}
```

```
// Подкласс для представления прямоугольников,
// производный от класса TwoDShape
class Rectangle extends TwoDShape {
   // Конструктор по умолчанию
   Rectangle() {
      super();
   }

   // Конструктор класса Rectangle
   Rectangle(double w, double h) {
      super(w, h, "прямоугольник"); // вызов конструктора суперкласса
   }

   // Создание квадрата
   Rectangle(double x) {
      super(x, "прямоугольник"); // вызов конструктора суперкласса
   }

   // Создание одного объекта на основе другого
   Rectangle(Rectangle ob) {
      super(ob); // передача объекта конструктору класса TwoDShape
   }

   boolean isSquare() {
      if(getWidth() == getHeight()) return true;
      return false;
   }

   // Переопределение метода area() для класса Rectangle
   double area() { ◄──── Переопрелеление метода area()
                          для класса Reactangle
      return getWidth() * getHeight();
   }
}

class DynShapes {
   public static void main(String args[]) {
      TwoDShape shapes[] = new TwoDShape[5];

      shapes[0] = new Triangle("контурный", 8.0, 12.0);
      shapes[1] = new Rectangle(10);
      shapes[2] = new Rectangle(10, 4);
      shapes[3] = new Triangle(7.0);

      shapes[4] = new TwoDShape(10, 20, "фигура");

      for(int i=0; i < shapes.length; i++) {
         System.out.println("Объект - " + shapes[i].getName());
         System.out.println("Площадь - " + shapes[i].area()); ◄──
                  Для каждой фигуры вызывается соответствующий метод area()
         System.out.println();
      }
   }
}
```

Ниже приведен результат выполнения данной программы.

```
Объект - треугольник
Площадь - 48.0

Объект - прямоугольник
Площадь - 100.0

Объект - прямоугольник
Площадь - 40.0

Объект - треугольник
Площадь - 24.5

Объект - фигура
Метод area() должен быть переопределен
Площадь - 0.0
```

Рассмотрим код данной программы более подробно. Теперь, как и предполагалось при написании программы, метод `area()` входит в состав класса `TwoDShape` и переопределяется в классах `Triangle` и `Rectangle`. В классе `TwoDShape` метод `area()` играет роль заполнителя и лишь уведомляет пользователя о том, что этот метод должен быть переопределен в подклассе. При каждом переопределении метода `area()` в нем реализуются средства, необходимые для того типа объекта, который инкапсулируется в подклассе. Так, если требуется реализовать класс для эллипсов, метод `area()` придется переопределить таким образом, чтобы он вычислял площадь этой фигуры.

Рассматриваемая здесь программа имеет еще одну важную особенность. Обратите внимание на то, что в методе `main()` геометрические фигуры объявляются как массив объектов типа `TwoDShape`. Но на самом деле элементами массива являются ссылки на объекты `Triangle`, `Rectangle` и `TwoDShape`. Это вполне допустимо. Ведь, как пояснялось ранее, ссылочная переменная суперкласса может ссылаться на объект его подкласса. В этой программе организован перебор элементов массива в цикле и вывод сведений о каждом объекте. Несмотря на всю простоту данного примера, он наглядно демонстрирует потенциальные возможности как наследования классов, так и переопределения методов. Тип объекта, на который указывает ссылочная переменная суперкласса, определяется во время выполнения, что гарантирует правильный выбор версии переопределенного метода. Если объект является производным от класса `TwoDShape`, то для вычисления его площади достаточно вызвать метод `area()`. Интерфейс для выполнения данной операции оказывается общим и не зависит от того, с какой именно геометрической фигурой приходится иметь дело.

Использование абстрактных классов

Иногда требуется создать суперкласс, в котором определяется лишь самая общая форма для всех его подклассов, а наполнение ее деталями предоставляется каждому из этих подклассов. В таком классе определяется лишь суть методов, которые должны быть конкретно реализованы в подклассах, а не в самом суперклассе. Подобная ситуация возникает, например, в связи с невозможностью полноценной реализации метода в суперклассе. Именно такая ситуация была продемонстрирована в варианте класса `TwoDShape` из предыдущего примера, где метод `area()` был определен всего лишь как заполнитель. Такой метод не вычисляет и не выводит площадь двумерной геометрической формы любого типа.

Создавая собственные библиотеки классов, вы сможете сами убедиться в том, что ситуации, когда невозможно дать полное определение метода в контексте его суперкласса, встречаются довольно часто. Подобное затруднение разрешается двумя способами. Один из них, как было показано в предыдущем примере, состоит в том, чтобы просто вывести предупреждающее сообщение. И хотя в некоторых случаях, например при отладке, такой способ может быть действительно полезным, в практике программирования он обычно не применяется. Ведь в суперклассе могут быть объявлены методы, которые должны быть переопределены в подклассе, чтобы этот класс приобрел конкретный смысл. Рассмотрим для примера класс `Triangle`. Он был бы неполным, если бы в нем не был переопределен метод `area()`. В подобных случаях требуется какой-то способ, гарантирующий, что в подклассе действительно будут переопределены все необходимые методы. И такой способ имеется в Java: он состоит в использовании *абстрактного метода*.

Абстрактный метод объявляется с использованием спецификации `abstract`. Абстрактный метод не имеет тела и потому не реализуется в суперклассе. Это означает, что он должен быть переопределен в подклассе, поскольку его вариант из суперкласса просто непригоден для использования. Для определения абстрактного метода используется следующий общий синтаксис:

```
abstract тип имя(список_параметров);
```

Как видите, в этом синтаксисе отсутствует тело. Спецификация `abstract` может применяться только к обычным методам, но не к статическим или конструкторам.

Класс, содержащий один или несколько абстрактных методов, должен быть также объявлен как абстрактный с использованием той же спецификации `abstract` в объявлении класса. Поскольку абстрактный класс не определяет реализацию полностью, у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора `new` приведет к появлению ошибки во время компиляции.

Подкласс, наследующий абстрактный класс, должен реализовать все абстрактные методы суперкласса. В противном случае он также должен быть определен как абстрактный. Таким образом, атрибут abstract наследуется до тех пор, пока не будет достигнута полная реализация класса.

Используя абстрактный класс, мы можем усовершенствовать рассмотренный ранее класс TwoDShape. Для неопределенной двумерной геометрической фигуры понятие площади не имеет смысла, поэтому в приведенной ниже версии программы метод area() и сам класс TwoDShape объявляются как абстрактные. Это, конечно, означает, что любой класс, наследующий класс TwoDShape, должен переопределить метод area().

```
// Создание абстрактного класса
abstract class TwoDShape { ◄——— Класс TwoDShape теперь абстрактный
   private double width;
   private double height;
   private String name;

   // Конструктор по умолчанию
   TwoDShape() {
      width = height = 0.0;
      name = "none";
   }

   // Параметризированный конструктор
   TwoDShape(double w, double h, String n) {
      width = w;
      height = h;
      name = n;
   }

   // Создание объекта с одинаковыми значениями
   // переменных экземпляра width и height
   TwoDShape(double x, String n) {
      width = height = x;
      name = n;
   }

   // Создание одного объекта на основе другого
   TwoDShape(TwoDShape ob) {
      width = ob.width;
      height = ob.height;
      name = ob.name;
   }

   // Методы доступа к переменным width и height
   double getWidth() { return width; }
   double getHeight() { return height; }
   void setWidth(double w) { width = w; }
   void setHeight(double h) { height = h; }

   String getName() { return name; }
```

```
    void showDim() {
        System.out.println("Ширина и высота - " + width + " и " +
                           height);
    }

    // Теперь метод area() абстрактный
    abstract double area(); ◄——— Превращение area()
}                                 в абстрактный метод

// Подкласс для представления треугольников,
// производный от класса TwoDShape
class Triangle extends TwoDShape {
    private String style;

    // Конструктор по умолчанию
    Triangle() {
        super();
        style = "none";
    }

    // Конструктор класса Triangle
    Triangle(String s, double w, double h) {
        super(w, h, "треугольник");
        style = s;
    }

    // Конструктор с одним аргументом для построения треугольника
    Triangle(double x) {
        super(x, "треугольник"); // вызвать конструктор суперкласса
        style = "закрашенный";
    }

    // Создание одного объекта на основе другого
    Triangle(Triangle ob) {
        super(ob); // передача объекта конструктору класса TwoDShape
        style = ob.style;
    }

    double area() {
        return getWidth() * getHeight() / 2;
    }

    void showStyle() {
        System.out.println("Треугольник " + style);
    }
}

// Подкласс для представления прямоугольников,
// производный от класса TwoDShape
class Rectangle extends TwoDShape {
    // Конструктор по умолчанию
    Rectangle() {
```

```
      super();
   }

   // Конструктор класса Rectangle
   Rectangle(double w, double h) {
      super(w, h, "прямоугольник"); // вызвать конструктор суперкласса
   }

   // Создание квадрата
   Rectangle(double x) {
      super(x, "прямоугольник"); // вызвать конструктор суперкласса
   }

   // Создание одного объекта на основе другого
   Rectangle(Rectangle ob) {
      super(ob); // передача объекта конструктору класса TwoDShape
   }

   boolean isSquare() {
      if(getWidth() == getHeight()) return true;
      return false;
   }

   double area() {
      return getWidth() * getHeight();
   }
}

class AbsShape {
   public static void main(String args[]) {
      TwoDShape shapes[] = new TwoDShape[4];

      shapes[0] = new Triangle("контурный", 8.0, 12.0);
      shapes[1] = new Rectangle(10);
      shapes[2] = new Rectangle(10, 4);
      shapes[3] = new Triangle(7.0);

      for(int i=0; i < shapes.length; i++) {
         System.out.println("Объект - " + shapes[i].getName());
         System.out.println("Площадь - " + shapes[i].area());
         System.out.println();
      }
   }
}
```

Как видно из текста программы, все классы, наследующие класс TwoDShape, должны переопределять метод area(). Вы можете убедиться в этом самостоятельно, попытавшись создать подкласс, в котором метод area() не переопределен. В итоге вы получите сообщение об ошибке во время компиляции. Разумеется, возможность создания объектной ссылки типа TwoDShape, что и было сделано в приведенном выше примере программы, у вас остается, но объявить

объект типа TwoDShape вы уже не сможете. Именно поэтому массив shapes в методе main() уменьшен до четырех элементов, а объект типа TwoDShape для абстрактной двумерной геометрической фигуры больше не создается.

И еще одно, последнее замечание. Обратите внимание на то, что в классе TwoDShape по-прежнему определяются методы showDim() и getName() без модификатора abstract. Ничего предосудительного в этом нет, поскольку допускается (и это часто используется на практике), чтобы абстрактные классы включали в себя конкретные методы, к которым подклассы могут обращаться в своем исходном коде. Переопределению в подклассах подлежат лишь те методы, которые объявлены как abstract.

Использование ключевого слова final

Механизмы наследования классов и переопределения методов — весьма мощные средства Java, однако иногда они могут становиться для вас помехой. Предположим, например, что создается класс, в котором инкапсулированы средства управления некоторым устройством. Данный класс может предоставлять пользователю возможность инициализировать устройство с использованием конфиденциальной коммерческой информации. В таком случае пользователи данного класса не должны иметь возможность переопределять метод, ответственный за инициализацию устройства. Для этой цели в Java предусмотрено ключевое слово final, позволяющее без труда запретить переопределение метода или наследование класса.

Предотвращение переопределения методов

Для того чтобы предотвратить переопределение метода, в начале его объявления нужно указать спецификацию final. Переопределять объявленные указанным способом методы нельзя. Ниже приведен фрагмент кода, демонстрирующий использование ключевого слова final для подобных целей.

```
class A {
   final void meth() {
      System.out.println("Это метод final.");
   }
}

class B extends A {
   void meth() { // Ошибка: этот метод не может быть переопределен!
      System.out.println("Недопустимо!");
   }
}
```

Поскольку метод meth() объявлен как final, его нельзя переопределить в классе B. Если вы попытаетесь сделать это, возникнет ошибка при компиляции программы.

Предотвращение наследования

Предотвратить наследование класса можно, указав в определении класса ключевое слово `final`. В этом случае считается, что данное ключевое слово применяется ко всем методам класса. Очевидно, что не имеет никакого смысла применять ключевое слово `final` к абстрактным классам. Ведь абстрактный класс не завершен по определению, и объявленные в нем методы должны быть реализованы в подклассах.

Ниже приведен пример класса, создание подклассов которого запрещено.

```
final class A {
   // ...
}

// Следующее определение класса недопустимо
class B extends A { // Ошибка: класс A не может иметь подклассов!
   // ...
}
```

Как следует из комментариев к данному примеру, наследование классом `B` класса `A` запрещено, так как последний определен как `final`.

Применение ключевого слова `final` к переменным экземпляра

Помимо рассмотренных ранее примеров использования, ключевое слово `final` можно применять и к переменным экземпляра. Подобным способом создаются именованные константы. Если имени переменной предшествует спецификация `final`, то значение этой переменной не может быть изменено на протяжении всего времени выполнения программы. Очевидно, что подобным переменным нужно присваивать начальные значения. В главе 6 был рассмотрен простой класс `ErrorMsg` для обработки ошибок. В нем устанавливается соответствие между кодами ошибок и строками сообщений об ошибках. Ниже приведен усовершенствованный вариант этого класса, в котором для создания именованных констант применяется спецификация `final`. Теперь, вместо того чтобы передавать методу `getErrorMsg()` числовое значение, например 2, достаточно указать при его вызове именованную целочисленную константу `DISKERR`.

```
// Возврат объекта типа String
class ErrorMsg {
   // Коды ошибок
   final int OUTERR = 0;
   final int INERR = 1;   ◄——— Объявление констант final
   final int DISKERR = 2;
   final int INDEXERR = 3;

   String msgs[] = {
      "Ошибка вывода",
```

```
        "Ошибка ввода",
        "Отсутствует место на диске",
        "Выход индекса за границы диапазона"
    };

    // Возврат сообщения об ошибке
    String getErrorMsg(int i) {
        if(i >=0 & i < msgs.length)
            return msgs[i];
        else
            return "Несуществующий код ошибки";
    }
}

class FinalD {
    public static void main(String args[]) {
        ErrorMsg err = new ErrorMsg();

        System.out.println(err.getErrorMsg(err.OUTERR));
        System.out.println(err.getErrorMsg(err.DISKERR));
    }
}
```

Использование констант final

Обратите внимание на то, как используются константы в методе `main()`. Они являются членами класса `ErrorMsg`, и поэтому для доступа к ним требуется ссылка на объект этого класса. Разумеется, константы могут быть унаследованы подклассами и быть непосредственно доступными в них.

Многие программисты используют имена констант типа `final`, состоящие полностью из прописных букв, как это сделано в предыдущем примере. Но данное правило не является строгим и лишь отражает общепринятый стиль программирования.

Спросим у эксперта

ВОПРОС. Могут ли переменные типа `final` быть статическими? И можно ли использовать ключевое слово `final` при объявлении локальных переменных и параметров методов?

ОТВЕТ. Да, можно. Объявив константу таким образом, вы сможете обращаться к ней по имени класса, не создавая конкретных объектов. Так, если при объявлении констант в классе `ErrorMsg` указать ключевое слово `static`, то вызов метода `println()` в методе `main()` может быть таким.

```
System.out.println(err.getErrorMsg(ErrorMsg.OUTERR));
System.out.println(err.getErrorMsg(ErrorMsg.DISKERR));
```

Благодаря объявлению параметра как `final` предотвращается его изменение в методе. А если объявить локальную переменную как `final`, то ей нельзя будет присвоить значение больше одного раза.

Класс Object

В Java определен специальный класс `Object`, который по умолчанию считается суперклассом всех остальных классов. Иными словами, все классы являются подклассами, производными от класса `Object`. Это означает, что ссылочная переменная типа `Object` может ссылаться на объект любого класса. Более того, такая переменная также может ссылаться на массив, поскольку массивы реализованы в виде классов.

В классе `Object` определены перечисленные ниже методы, доступные в любом объекте.

Метод	Назначение
`Object clone()`	Создает новый объект, аналогичный клонируемому объекту
`boolean equals(Object` *объект*`)`	Определяет равнозначность объектов
`void finalize()`	Вызывается перед тем, как неиспользуемый объект будет удален сборщиком мусора (не рекомендуется в JDK 9)
`Class<?> getClass()`	Определяет класс объекта во время выполнения
`int hashCode()`	Возвращает хеш-код, связанный с вызывающим объектом
`void notify()`	Возобновляет работу потока, ожидающего уведомления от вызывающего объекта
`void notifyAll()`	Возобновляет работу всех потоков, ожидающих уведомления от вызывающего объекта
`String toString()`	Возвращает символьную строку, описывающую объект

`void wait()`
`void wait(long` *миллисекунды*`)`
`void wait(long` *миллисекунды*, `int` *наносекунды*`)`
	Ожидает выполнения другого потока

Методы `getClass()`, `notify()`, `notifyAll()` и `wait()` объявлены как `final`, а остальные можно переопределять в подклассах. Некоторые из этих методов будут описаны далее. Два из них — `equals()` и `toString()` — заслуживают особого внимания. Метод `equals()` сравнивает два объекта. Если объекты равнозначны, то он возвращает логическое значение `true`, иначе — логическое значение `false`. Метод `toString()` возвращает символьную строку, содержащую описание того объекта, которому принадлежит этот метод. Он автоматически вызывается в том случае, если объект передается методу `println()` в качестве параметра. Во многих классах этот метод переопределяется. В этом случае

описание специально подбирается для конкретных типов объектов, которые в них создаются.

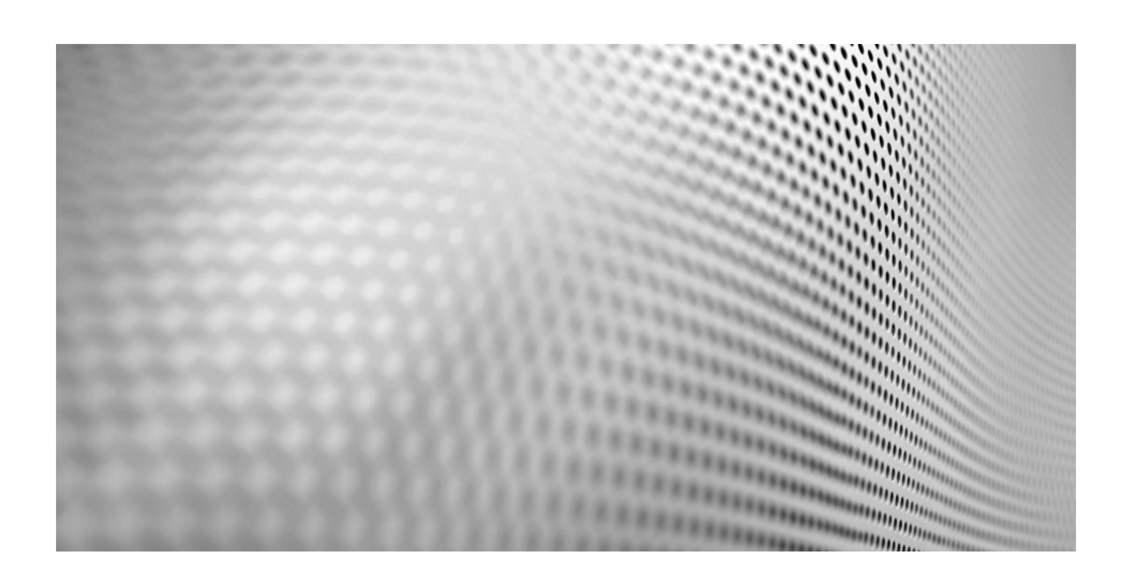
Обратите внимание на необычный синтаксис, описывающий значение, возвращаемое методом `getClass()`. Это *обобщенный тип*. С помощью обобщений в Java можно указывать в качестве параметра тип данных, используемый в классе или методе. Более подробно обобщения будут рассмотрены в главе 13.

Вопросы и упражнения для самопроверки

1. Имеет ли суперкласс доступ к членам подкласса? Имеет ли подкласс доступ к членам суперкласса?
2. Создайте подкласс `Circle`, производный от класса `TwoDShape`. В нем должен быть определен метод `area()`, вычисляющий площадь круга, а также конструктор с ключевым словом `super` для инициализации членов, унаследованных от класса `TwoDShape`.
3. Как предотвратить обращение к членам суперкласса из подкласса?
4. Опишите назначение и два варианта использования ключевого слова `super`.
5. Допустим, имеется следующая иерархия классов.

```
class Alpha { ...

class Beta extends Alpha { ...

Class Gamma extends Beta { ...
```

   В каком порядке вызываются конструкторы этих классов при создании объекта класса `Gamma`?
6. Переменная ссылки на суперкласс может указывать на объект подкласса. Объясните, почему это важно и как это связано с переопределением методов?
7. Что такое абстрактный класс?
8. Как предотвратить переопределение метода и наследование класса?
9. Объясните, каким образом механизмы наследования, переопределения методов и абстрактные классы используются для поддержки полиморфизма.
10. Какой класс является суперклассом для всех остальных классов?
11. Класс, который содержит хотя бы один абстрактный метод, должен быть объявлен абстрактным. Верно или не верно?
12. Какое ключевое слово следует использовать для создания именованной константы?



Глава 8

Пакеты и интерфейсы

В этой главе...

- Использование пакетов
- Влияние пакетов на доступ к членам класса
- Использование модификатора доступа `protected`
- Импорт пакетов
- Стандартные пакеты Java
- Основные сведения об интерфейсах
- Реализация интерфейсов
- Использование интерфейсных ссылок
- Переменные интерфейса
- Наследование интерфейсов
- Создание методов по умолчанию, статических и закрытых методов интерфейсов

Эта глава посвящена двум очень важным инновационным средствам Java: пакетам и интерфейсам. *Пакет* — это группа логически связанных классов. Пакеты помогают лучше организовать код и обеспечивают дополнительный уровень инкапсуляции. Как будет продемонстрировано в главе 15, пакеты также играют важную роль при создании и использовании модулей — нового средства, которое появилось в JDK 9. *Интерфейс* определяет набор методов, которые должны предоставляться в классе. В самом интерфейсе эти методы не реализуются, лишь анонсируются. Пакеты и интерфейсы предлагают дополнительные возможности для более рациональной организации программ и контроля их структуры.

Пакеты

Иногда взаимозависимые части программ удобно объединять в группы. В Java для этой цели предусмотрены пакеты. Прежде всего, пакет предоставляет механизм объединения взаимосвязанных частей программы. При обращении к классам, входящим в пакет, указывается его имя. Таким образом, пакеты дают возможность именовать коллекции классов. И кроме того, пакет является частью механизма управления доступом в Java. Классы могут быть объявлены как закрытые для всех пакетов, исключая тот, в который они входят. Следовательно, пакет обеспечивает также средства для инкапсуляции классов. Рассмотрим все эти средства более подробно.

При именовании класса для него выделяется имя в *пространстве имен*. При этом пространство имен определяет область объявлений. В Java не допускается присваивание двум классам одинаковых имен из одного и того же пространства имен. Иными словами, в пределах пространства имен каждый класс должен обладать уникальным именем. В примерах программ, представленных в предыдущих главах, по умолчанию использовалось глобальное пространство имен. Это удобно для небольших программ, но по мере увеличения объема кода могут возникать конфликты имен. В крупных программах бывает нелегко выбрать уникальное имя для класса. Более того, при использовании библиотек и кода, написанного другими программистами, приходится принимать специальные меры, чтобы предотвратить конфликт имен. Для разрешения подобных затруднений служат пакеты, позволяющие разделить пространство имен на отдельные области. Если класс определен в пакете, то имя пакета присоединяется к имени класса, в результате чего исключается конфликт между двумя классами с одинаковыми именами, но принадлежащими к разным пакетам.

Пакет обычно содержит логически связанные классы, и поэтому в Java определены специальные права доступа к содержимому пакета. Так, в пакете можно определить код, доступный другому коду из того же самого пакета, но недоступный из других пакетов. Это позволяет создавать автономные группы связанных классов и присваивать операциям, выполняемым в этих пакетах, статус закрытых.

Определение пакета

Каждый класс в Java относится к тому или иному пакету. Если инструкция `package` отсутствует в коде, то используется глобальный пакет, выбираемый по умолчанию. Пакет по умолчанию не обладает именем, что упрощает его применение. Именно поэтому в рассмотренных до сих пор примерах программ не нужно было беспокоиться о пакетах. Но пакет по умолчанию подходит только для очень простых программ, служащих в качестве примера, тогда как для реальных приложений он малопригоден. Как правило, для разрабатываемого кода приходится определять один или несколько пакетов.

Чтобы создать пакет, достаточно поместить инструкцию `package` в начало файла, содержащего исходный код программы на Java. В результате классы, определенные в этом файле, будут принадлежать указанному пакету. А поскольку пакет определяет пространство имен, имена классов, содержащихся в файле, войдут в это пространство имен как его составные части.

Общая форма инструкции `package` такова:

```
package имя_пакета;
```

Например, приведенная ниже строка кода определяет пакет `mypack`.

```
package mypack;
```

Для управления пакетами в Java используется файловая система, в которой с целью хранения содержимого каждого пакета выделяется отдельный каталог. Например, файлы с расширением `.class`, содержащие классы и объявленные в пакете `mypack`, будут храниться в каталоге `mypack`.

Подобно другим именам в Java, имена пакетов зависят от регистра символов. Это означает, что каталог, предназначенный для хранения пакета, должен обладать именем, в точности совпадающим с названием пакета. Если у вас возникнут затруднения при опробовании примеров программ, представленных в этой главе, проверьте соответствие имен пакетов именам каталогов. Пакеты всегда именуются прописными буквами.

В разных файлах могут содержаться одинаковые инструкции `package`. Эта инструкция лишь определяет, к какому именно пакету должны принадлежать классы, код которых содержится в данном файле, и не запрещает другим классам входить в состав того же самого пакета. Как правило, пакеты реальных программ распространяются на большое количество файлов.

В Java допускается создавать иерархию пакетов. Для этого достаточно разделить имена пакетов точками. Ниже приведена общая форма инструкции `package` для определения многоуровневого пакета.

```
package пакет_1.пакет_2.пакет_3...пакет_N;
```

Само собой разумеется, что для поддержки иерархии пакетов следует создать аналогичную иерархию каталогов.

```
package alpha.beta.gamma;
```

Классы, содержащиеся в данном пакете, должны храниться в структуре каталогов `.../alpha/beta/gamma`, где многоточие обозначает путь к каталогу `alpha`.

Поиск пакетов и переменная среды CLASSPATH

Как уже говорилось выше, иерархия каталогов пакетов должна отражать иерархию самих пакетов. В связи с этим возникает интересный вопрос: как исполняющая среда Java узнает, где искать созданные пакеты? Ответ на этот вопрос состоит из трех частей. Во-первых, по умолчанию исполняющая среда обращается к текущему рабочему каталогу. Так, если поместить пакет в подкаталоге текущего каталога, он будет там найден. Во-вторых, один или несколько путей к каталогам можно задать в качестве значения переменной среды `CLASSPATH`. И в-третьих, при вызове интерпретатора `java` и компилятора `javac` из командной строки можно указать параметр `-classpath`, а также путь к каталогам с классами.

Рассмотрим в качестве примера следующее определение пакета:

```
package mypack
```

Для того чтобы программа могла найти пакет `mypack`, должно быть выполнено одно из трех условий: программа должна быть запущена из каталога,

содержащего пакет mypack; в переменной среды CLASSPATH должен храниться полный путь к каталогу с пакетом mypack; при запуске программы интерпретатору java должен быть передан параметр -classpath и указан путь к каталогу с пакетом mypack.

Вам будет проще всего работать с примерами программ, представленными далее, если вы создадите в текущем каталоге, используемом для разработки программ, структуру каталогов, предназначенных для хранения пакетов, а затем разместите файлы с расширением .class в соответствующих каталогах. Далее следует выполнять программы из каталога, выбранного для разработки.

Во избежание каких-либо осложнений все файлы с расширениями .java и .class рекомендуется хранить в том каталоге, который выделен для содержащего их пакета. Кроме того, каждый исходный файл следует компилировать из того каталога, который находится по иерархии выше каталога с пакетом.

Простой пример применения пакета

С учетом всего вышесказанного напишем небольшую программу, использующую пакет, а затем запустим ее на выполнение. Эта программа предназначена для создания простой базы данных книг. Классы этой программы будут содержаться в пакете bookpack.

```
// Простая программа, демонстрирующая применение пакетов
package bookpack;   ◄——— Этот файл является частью пакета bookpack

class Book {   ◄——— Таким образом, класс Book является частью пакета bookpack
   private String title;
   private String author;
   private int pubDate;

   Book(String t, String a, int d) {
      title = t;
      author = a;
      pubDate = d;
   }

   void show() {
      System.out.println(title);
      System.out.println(author);
      System.out.println(pubDate);
      System.out.println();
   }
}                         Этот класс также входит в пакет bookpack

class BookDemo {   ◄———┘
   public static void main(String args[]) {
      Book books[] = new Book[5];

      books[0] = new Book("Java: руководство для начинающих,
                          7-е издание", "Герберт Шилдт", 2018);
```

```
        books[1] = new Book("Java: полное руководство, 10-е издание",
                            "Герберт Шилдт", 2018);
        books[2] = new Book("Искусство программирования на Java",
                            "Герберт Шилдт", 2005);
        books[3] = new Book("Красный шторм поднимается",
                            "Том Клэнси", 2006);
        books[4] = new Book("В дороге ", "Джек Керуак", 2012);

        for(int i=0; i < books.length; i++) books[i].show();
    }
}
```

Присвойте файлу с приведенным выше исходным кодом имя `BookDemo.java` и поместите его в каталог `bookpack`.

Скомпилируйте этот файл, введя в командной строке следующую команду:

```
javac bookpack/BookDemo.java
```

После этого попробуйте выполнить скомпилированную программу, введя в командной строке такую команду:

```
java bookpack.BookDemo
```

Не забывайте, что для нормального выполнения указанных выше команд текущим должен быть каталог, являющийся родительским по отношению к каталогу `bookpack`. (Для компиляции и запуска программы из какого-нибудь другого каталога вам придется указать путь к каталогу `bookpack`, используя один из двух других описанных выше способов обращения к каталогам с пакетами.)

Теперь классы `BookDemo` и `Book` относятся к пакету `bookpack`. Это означает, что при вызове интерпретатора нельзя ограничиваться передачей ему только имени класса `BookDemo`. Приведенная ниже команда не будет выполнена:

```
java BookDemo
```

Перед именем класса `BookDemo` следует указать имя его пакета, как показано выше.

Пакеты и доступ к членам классов

В предыдущих главах были представлены основные механизмы управления доступом, в том числе модификаторы `private` и `public`. И теперь самое время продолжить обсуждение вопросов управления доступом к членам классов, ведь и пакеты принимают участие в управлении доступом. Прежде чем продолжить рассмотрение материала, следует отметить, что модули, которые появились в JDK 9, предлагают новые способы обеспечения доступа, но мы сосредоточимся исключительно на способах, предусматривающих взаимодействие пакетов и классов.

Область действия члена класса определяется используемым модификатором доступа: `private`, `public` или `protected`, хотя модификатор может и отсутствовать. На формирование области действия оказывает также влияние

принадлежность класса к тому или иному пакету. Таким образом, область действия члена класса определяется его доступностью как в классе, так и в пакете. Столь сложный, многоуровневый подход к управлению доступом позволяет установить достаточно обширный набор прав доступа. В табл. 8.1 описаны разные уровни доступа к членам классов. Рассмотрим каждый из них в отдельности.

Таблица 8.1. Уровни доступа к членам классов

	Закрытый член	Член, доступный по умолчанию	Защищенный член	Открытый член
Доступен в том же классе	Да	Да	Да	Да
Доступен из подкласса в том же пакете	Нет	Да	Да	Да
Доступен из любого класса в том же пакете	Нет	Да	Да	Да
Доступен из подкласса в любом пакете	Нет	Нет	Да	Да
Доступен из всякого класса в любом пакете	Нет	Нет	Нет	Да

Если модификатор доступа явно не указан для члена класса, то он доступен только в своем пакете, но не за его пределами. Следовательно, член класса, для которого не задан модификатор доступа, является открытым в текущем пакете и закрытым за его пределами.

Члены класса, объявленные как открытые (`public`), доступны из классов, принадлежащих любым пакетам. На доступ к ним никаких ограничений не накладывается. А члены класса, объявленные как закрытые (`private`), доступны только для членов того же самого класса. Другие классы, даже принадлежащие к тому же самому пакету, не могут воздействовать на них. И наконец, члены класса, объявленные как защищенные (`protected`), доступны для классов, находящихся в том же самом пакете, а также для подклассов данного класса, независимо от того, каким пакетам эти подклассы принадлежат.

Правила доступа, приведенные в табл. 8.1, распространяются только на члены классов. Сами же классы могут быть объявлены как открытые или доступные по умолчанию. Если в определении класса стоит ключевое слово `public`, то он доступен для других классов. Отсутствие модификатора доступа означает, что класс доступен только для классов, находящихся в том же пакете. На классы, объявленные как открытые, накладывается следующее единственное ограничение: имя файла, в котором находится исходный код класса, должно совпадать с именем класса.

Примечание

Модули, которые появились в JDK 9, также влияют на способы доступа. Модули будут подробно рассмотрены в главе 15.

Пример доступа к пакету

В рассмотренном выше примере классы Book и BookDemo находились в одном и том же пакете, поэтому при организации доступа из класса BookDemo к классу Book не возникало никаких затруднений. По умолчанию все члены класса имеют право обращаться к членам других классов из того же самого пакета. Если бы класс Book находился в одном пакете, а класс BookDemo — в другом, то ситуация оказалась бы немного сложнее. В этом случае доступ к классу Book по умолчанию был бы запрещен. Для того чтобы сделать класс Book доступным для других пакетов, в код программы нужно внести три изменения. Во-первых, сам класс Book должен быть объявлен открытым (public). Это позволит обращаться к нему за пределами пакета bookpack. Во-вторых, конструктор класса должен быть также объявлен открытым. И в-третьих, модификатор доступа public следует указать также перед методом show(). Благодаря этому конструктор и метод show() станут доступными за пределами пакета bookpack. Следовательно, для использования класса Book в классах, принадлежащих другим пакетам, его следует объявить так, как показано ниже.

```
// Класс Book, видоизмененный для открытого доступа
package bookpack;

public class Book {  ◄—— Класс Book и его члены должны быть объявлены открытыми,
                          чтобы их можно было использовать в других пакетах
   private String title;
   private String author;
   private int pubDate;

   // Теперь конструктор стал открытым
   public Book(String t, String a, int d) {
      title = t;
      author = a;
      pubDate = d;
   }

   // Теперь метод стал открытым
   public void show() {
      System.out.println(title);
      System.out.println(author);
      System.out.println(pubDate);
      System.out.println();
   }
}
```

Для того чтобы воспользоваться классом Book в другом пакете, нужно применить инструкцию import, которая рассматривается в следующем разделе,

либо указать полное имя класса, т.е. предварить имя класса именем пакета. Ниже приведен пример класса UseBook, содержащегося в пакете bookpackext. Для обращения к классу Book в нем используется полное имя этого класса.

```
// Данный класс принадлежит пакету bookpackext
package bookpackext;

// Использование класса Book из пакета bookpack
class UseBook {
   public static void main(String args[]) {
      bookpack.Book books[] = new bookpack.Book[5];
      books[0] = new bookpack.Book("Java: руководство для начинающих,
                                   7-е издание", "Герберт Шилдт",
                                   2018);
      books[1] = new bookpack.Book("Java: полное руководство,
                                   10-е издание", "Герберт Шилдт",
                                   2018);
      books[2] = new bookpack.Book("Искусство программирования на
                                   Java", "Герберт Шилдт",
                                   2005);
      books[3] = new bookpack.Book("Красный шторм поднимается",
                                   "Том Клэнси", 2006);
      books[4] = new bookpack.Book("В дороге", "Джек Керуак", 2012);

      for(int i=0; i < books.length; i++) books[i].show();
   }
}
```

Перед именем класса Book указывается имя пакета bookpack

Обратите внимание на то, что при каждом обращении к классу Book перед ним указывается имя пакета bookpack. Если бы здесь не использовалось полное имя, то при компиляции класса UseBook класс Book не был бы найден.

Защищенные члены классов

Начинающие программисты иногда неправильно пользуются модификатором доступа protected. Как пояснялось ранее, переменные и методы, объявленные защищенными (protected), доступны для классов, находящихся в том же самом пакете, а также для подклассов данного класса, независимо от того, каким пакетам они принадлежат. Иными словами, член класса, объявленный как protected, доступен для подклассов, но защищен от доступа за пределами пакета.

Для того чтобы стало понятнее назначение модификатора доступа protected, рассмотрим следующий пример. Сначала изменим класс Book, объявив его переменные экземпляра защищенными.

```
// Объявление защищенными переменных экземпляра в классе Book
package BookPack;

public class Book {
   // При объявлении этих переменных использован
```

```
   // модификатор доступа protected
   protected String title;
   protected String author;      ← Теперь эти переменные объявлены как protected
   protected int pubDate;

   public Book(String t, String a, int d) {
      title = t;
      author = a;
      pubDate = d;
   }

   public void show() {
      System.out.println(title);
      System.out.println(author);
      System.out.println(pubDate);
      System.out.println();
   }
}
```

После этого создадим подкласс ExtBook класса Book, а также класс ProtectDemo, в котором будет использоваться класс ExtBook. В классе ExtBook содержится поле, предназначенное для хранения названия издательства, а также несколько методов доступа. Оба класса принадлежат к пакету bookpackext. Их исходный код приведен ниже.

```
// Пример использования модификатора protected
package bookpackext;

class ExtBook extends bookpack.Book {
   private String publisher;

   public ExtBook(String t, String a, int d, String p) {
      super(t, a, d);
      publisher = p;
   }

   public void show() {
      super.show();
      System.out.println(publisher);
      System.out.println();
   }

   public String getPublisher() { return publisher; }
   public void setPublisher(String p) { publisher = p; }

   // Следующие инструкции допустимы, поскольку подклассы имеют
   // право доступа к членам класса, объявленным защищенными
   public String getTitle() { return title; }
   public void setTitle(String t) { title = t; }
   public String getAuthor() { return author; }  ← Доступ к членам класса Book
   public void setAuthor(String a) { author = a; }   разрешен для подклассов
```

```
   public int getPubDate() { return pubDate; }
   public void setPubDate(int d) { pubDate = d; }
}

class ProtectDemo {
   public static void main(String args[]) {
      ExtBook books[] = new ExtBook[5];

      books[0] = new ExtBook("Java: руководство для начинающих,
                             7-е издание", "Герберт Шилдт", 2018,
                             "Вильямс");
      books[1] = new ExtBook("Java: полное руководство,
                             10-е издание", "Герберт Шилдт", 2018,
                             "Вильямс");
      books[2] = new ExtBook("Искусство программирования на Java",
                             "Герберт Шилдт", 2003, "Диалектика");
      books[3] = new ExtBook("Красный шторм поднимается",
                             "Том Клэнси", 2006, "Эксмо");
      books[4] = new ExtBook("В дороге", "Джек Керуак", 2012,
                             "Азбука");

      for(int i=0; i < books.length; i++) books[i].show();

      // Поиск книг по автору
      System.out.println("Все книги Герберта Шилдта.");
      for(int i=0; i < books.length; i++)
         if(books[i].getAuthor() == "Герберт Шилдт")
            System.out.println(books[i].getTitle());

//    books[0].title = "test title"; // Ошибка: доступ запрещен!
   }
}
```

Доступ к защищенным полям класса Book возможен только из его подклассов

Обратите внимание на код класса ExtBook. В связи с тем что класс ExtBook является подклассом, производным от класса Book, он имеет доступ к защищенным членам класса Book. Это правило действует, несмотря на то что класс ExtBook находится в другом пакете. Следовательно, он может обращаться непосредственно к переменным экземпляра title, author и pubDate, что и было использовано при написании методов доступа. В то же время доступ к этим переменным экземпляра из класса ProtectDemo запрещен, поскольку класс ProtectDemo не является подклассом, производным от класса Book. Так, если удалить комментарии в приведенной ниже строке кода, то программа не будет скомпилирована.

```
//    books[0].title = "test title"; // Ошибка: доступ запрещен!
```

Импорт пакетов

При использовании класса из другого пакета необходимо задавать его полное имя, т.е. указывать перед именем класса имя пакета. Такой подход был применен в предыдущем примере. Но его соблюдение очень быстро становится утомительным для программирования, и особенно это касается глубоко вложенных пакетов. Язык Java был разработан программистами для программистов, и поэтому не удивительно, что в нем было предусмотрено более удобное средство доступа к содержимому пакета: инструкция `import`. Используя эту инструкцию, можно упростить обращение к одному или нескольким членам пакета, чтобы пользоваться ими непосредственно, не указывая явно имя пакета.

Ниже приведена общая форма инструкции `import`.

```
import имя_пакета.имя_класса;
```

Здесь *имя_пакета* — название пакета, которое может включать полный путь к пакету, а *имя_класса* — название импортируемого класса. Если нужно импортировать все содержимое пакета, вместо имени класса следует указать звездочку (*). Ниже приведены примеры обеих форм записи инструкции `import`.

```
import mypack.MyClass
import mypack.*;
```

В первом случае из пакета `mypack` импортируется класс `MyClass`, а во втором — все классы из данного пакета. В исходном файле программы на Java инструкции `import` должны следовать сразу же после инструкции `package` (если таковая имеется) и перед определением классов.

С помощью инструкции `import` можно организовать доступ к пакету `bookpack` и воспользоваться классом `Book`, не прибегая к его полному имени. Инструкция `import`, разрешающая данное затруднение, помещается в начало того файла, где требуется доступ к классу `Book`, в следующем виде:

```
import bookpack.*;
```

Например, так будет выглядеть исходный код класса `UseBook`, в котором используется механизм импорта пакетов.

```
// Использование ключевого слова import
package bookpackext;
import bookpack.*;   ◄——— Импорт класса bookpack

// Использование класса Book из пакета bookpack
class UseBook {
   public static void main(String args[]) {
      Book books[] = new Book[5]; ◄——— Теперь к классу Book можно обращаться
                                        непосредственно, без указания его
                                        полного имени

      books[0] = new Book("Java: руководство для начинающих,
                          7-е издание", "Герберт Шилдт", 2018);
      books[1] = new Book("Java: полное руководство,
                          10-е издание", "Герберт Шилдт", 2018);
```

```
        books[2] = new Book("Искусство программирования на Java",
                            "Герберт Шилдт", 2005);
        books[3] = new Book("Красный шторм поднимается",
                            "Том Клэнси", 2006);
        books[4] = new Book("В дороге", "Джек Керуак", 2012);

        for(int i=0; i < books.length; i++) books[i].show();
    }
}
```

Как видите, теперь нет нужды предварять имя класса `Book` именем пакета.

Библиотечные классы Java, содержащиеся в пакетах

Как пояснялось ранее, в Java определено большое количество стандартных классов, доступных всем программам. Библиотека классов Java обычно называется Java API (Application Programming Interface — прикладной программный интерфейс). Классы, входящие в состав Java API, хранятся в пакетах. На верхней ступени иерархии находится пакет `java`. В его состав входят подчиненные пакеты, включая и перечисленные ниже.

Пакет	Описание
`java.lang`	Содержит большое количество классов общего назначения
`java.io`	Содержит классы, предназначенные для поддержки ввода-вывода
`java.net`	Содержит классы, предназначенные для поддержки сетевого взаимодействия
`java.applet`	Содержит классы, предназначенные для создания аплетов
`java.awt`	Содержит классы, обеспечивающие поддержку набора инструментальных средств Abstract Window Toolkit (AWT)

В примерах программ, представленных в этой книге, с самого начала использовался пакет `java.lang`. Помимо прочего, он содержит класс `System` (к нему не раз приходилось обращаться при вызове метода `println()`). Пакет `java.lang` примечателен тем, что он автоматически включается в каждую программу на Java, в то время как содержимое других пакетов приходится импортировать явным образом. Некоторые из этих пакетов будут рассмотрены в последующих главах.

Интерфейсы

Иногда в объектно-ориентированном программировании полезно определить, что именно должен делать класс, но не то, как он должен это делать. Примером может служить упоминавшийся ранее абстрактный метод. В абстрактном

методе определяются возвращаемый тип и сигнатура метода, но не предоставляется его реализация. А в подклассе должна быть обеспечена своя собственная реализация каждого абстрактного метода, определенного в его суперклассе. Таким образом, абстрактный метод определяет *интерфейс*, но не *реализацию* метода. Конечно, абстрактные классы и методы приносят известную пользу, но положенный в их основу принцип может быть развит далее. В Java предусмотрено разделение интерфейса класса и его реализации с помощью ключевого слова `interface`.

С точки зрения синтаксиса интерфейсы подобны абстрактным классам. Но в интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Для реализации интерфейса в классе должны быть предоставлены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода в определении деталей своей собственной реализации интерфейса. Следовательно, один и тот же интерфейс может быть реализован в двух классах по-разному. Тем не менее в каждом из них должен поддерживаться один и тот же ряд методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым. Благодаря поддержке интерфейсов в Java может быть в полной мере реализован главный принцип полиморфизма: "один интерфейс — множество методов".

Прежде чем продолжить изучение материала, сделаем одно важное замечание. В JDK 8 интерфейсы существенно изменились. В версиях, предшествующих JDK 8, интерфейс не мог определять какую-либо реализацию, т.е. в этих версиях интерфейс мог определять, что делать, но не как делать, как было только что отмечено. В JDK 8 все изменилось, и теперь можно добавить в метод интерфейса *реализацию по умолчанию*. Более того, теперь поддерживаются методы статического интерфейса, а начиная с версии JDK 9 интерфейс может также включать закрытые методы. Все это привело к тому, что интерфейс может проявлять некоторое поведение. Но подобные методы по сути являются средствами специального назначения, а исходное предназначение интерфейса остается без изменений. Поэтому, как правило, вы по-прежнему будете часто создавать и использовать интерфейсы, в которых не будут применяться эти новые средства. Поэтому мы начнем рассматривать интерфейсы в их традиционной форме, а новые средства интерфейсов будут описаны в конце этой главы.

Интерфейсы объявляются с помощью ключевого слова `interface`. Ниже приведена упрощенная форма объявления интерфейса.

```
доступ interface имя {
   возвращаемый_тип имя_метода_1(список_параметров);
```

```
    возвращаемый_тип имя_метода_2(список_параметров);
    тип переменная_1 = значение;
    тип переменная_2 = значение;
    // ...
    возвращаемый_тип имя_метода_N(список_параметров);
    тип переменная_N = значение;

}
```

Здесь *доступ* обозначает тип доступа, который определяется модификатором доступа `public` или вообще не указывается. Если модификатор доступа отсутствует, применяется правило, предусмотренное по умолчанию, т.е. интерфейс считается доступным только членам своего пакета. Ключевое слово `public` указывает на то, что интерфейс может использоваться в любом другом пакете. (Код интерфейса, объявленного как `public`, должен храниться в файле, имя которого совпадает с именем интерфейса.) А имя интерфейса может быть любым допустимым идентификатором.

При объявлении методов указываются их сигнатуры и возвращаемые типы. Эти методы являются, по сути, абстрактными. Как упоминалось выше, реализация метода не может содержаться в составе интерфейса. Каждый класс, в определении которого указан интерфейс, должен реализовать все методы, объявленные в интерфейсе. Методы, объявленные в интерфейсе, неявно считаются открытыми (`public`).

Переменные, объявленные в интерфейсе, не являются переменными экземпляра. Они неявно обозначаются ключевыми словами `public`, `final` и `static` и обязательно подлежат инициализации. По сути, они являются константами. Ниже приведен пример определения интерфейса. Предполагается, что этот интерфейс должен быть реализован в классе, где формируется последовательный ряд числовых значений.

```
public interface Series {
    int getNext(); // возврат следующего по порядку числа
    void reset();  // сброс
    void setStart(int x); // установка начального значения
}
```

Этот интерфейс объявляется открытым (`public`), а следовательно, может быть реализован в классе, принадлежащем любому пакету.

Реализация интерфейсов

Определенный однажды интерфейс может быть реализован одним или несколькими классами. Для реализации интерфейса в объявление класса следует ввести ключевое слово `implements`, а затем определить методы, объявленные в интерфейсе. Ниже приведена общая форма реализации интерфейса в классе.

```
class имя_класса extends суперкласс implements интерфейс {
    // тело класса
}
```

Если в классе должно быть реализовано несколько интерфейсов, то их названия указываются через запятую. Разумеется, ключевое слово extends и имя суперкласса указывать не обязательно.

Реализуемые методы интерфейса должны быть объявлены открытыми (public). А сигнатура реализованного метода должна полностью соответствовать сигнатуре, объявленной в составе интерфейса. Ниже приведен пример класса ByTwos, реализующего рассмотренный ранее интерфейс Series. В этом классе формируется последовательный ряд числовых значений, каждое из которых на два больше предыдущего.

```
// Реализация интерфейса Series
class ByTwos implements Series {
   int start;
   int val;
                  Реализация интерфейса Series
   ByTwos() {
      start = 0;
      val = 0;
   }

   public int getNext() {
      val += 2;
      return val;
   }

   public void reset() {
      start = 0;
      val = 0;
   }

   public void setStart(int x) {
      start = x;
      val = x;
   }
}
```

Обратите внимание на то, что методы getNext(), reset() и setStart() объявлены открытыми. Это нужно сделать непременно, поскольку любой метод интерфейса неявно считается открытым для доступа. Ниже приведен пример программы, демонстрирующий применение класса ByTwos.

```
class SeriesDemo {
   public static void main(String args[]) {
      ByTwos ob = new ByTwos();

      for(int i=0; i < 5; i++)
         System.out.println("Следующее значение: " +
                            ob.getNext());

      System.out.println("\nСброс");
      ob.reset();
```

```
        for(int i=0; i < 5; i++)
            System.out.println("Следующее значение: " +
                                ob.getNext());

        System.out.println("\nНачальное значение: 100");
        ob.setStart(100);
        for(int i=0; i < 5; i++)
            System.out.println("Следующее значение: " +
                                ob.getNext());
    }
}
```

Выполнение этой программы дает следующий результат.

```
Следующее значение: 2
Следующее значение: 4
Следующее значение: 6
Следующее значение: 8
Следующее значение: 10

Сброс
Следующее значение: 2
Следующее значение: 4
Следующее значение: 6
Следующее значение: 8
Следующее значение: 10

Начальное значение: 100
Следующее значение: 102
Следующее значение: 104
Следующее значение: 106
Следующее значение: 108
Следующее значение: 110
```

Класс, реализующий интерфейс, может содержать дополнительные переменные и методы, что вполне допустимо. Более того, именно так в большинстве случаев и поступают те, кто программирует на Java. Например, в приведенную ниже версию класса ByTwos добавлен метод getPrevious(), возвращающий предыдущее числовое значение.

```
// Реализация интерфейса Series и добавление метода getPrevious()
class ByTwos implements Series {
    int start;
    int val;
    int prev;

    ByTwos() {
        start = 0;
        val = 0;
        prev = -2;
    }

    public int getNext() {
        prev = val;
```

```
        val += 2;
        return val;
    }

    public void reset() {
        start = 0;
        val = 0;
        prev = -2;
    }

    public void setStart(int x) {
        start = x;
        val = x;
        prev = x - 2;
    }

    int getPrevious() {  ◄——— Добавление метода, который не определен
        return prev;           в интерфейсе Series
    }
}
```

Обратите внимание на то, что для добавления метода `getPrevious()` пришлось изменить реализацию методов, объявленных в интерфейсе `Series`. Но сам интерфейс не претерпел никаких изменений. Эти изменения не видны за пределами класса и не влияют на его использование. В этом и состоит одно из преимуществ интерфейсов.

Как пояснялось ранее, интерфейс может быть реализован каким угодно количеством классов. В качестве примера ниже приведен код класса `ByThrees`, формирующего последовательный ряд числовых значений, каждое из которых на три больше предыдущего.

```
// Еще одна реализация интерфейса Series
class ByThrees implements Series {  ◄——— Другая реализация интерфейса Series
    int start;
    int val;

    ByThrees() {
        start = 0;
        val = 0;
    }

    public int getNext() {
        val += 3;
        return val;
    }

    public void reset() {
        start = 0;
        val = 0;
    }
```

```
    public void setStart(int x) {
        start = x;
        val = x;
    }
}
```

Следует также иметь в виду, что если в определении класса имеется ключевое слово `implements`, но он не реализует все методы указанного интерфейса, то этот класс должен быть объявлен абстрактным (`abstract`). Объект такого класса создать нельзя, но можно использовать его в качестве суперкласса, а завершить реализацию методов интерфейса — в его подклассах.

Применение интерфейсных ссылок

Возможно, вы будете несколько удивлены, узнав, что в Java допускается объявлять переменные ссылочного интерфейсного типа, т.е. переменные, хранящие ссылки на интерфейс. Такая переменная может ссылаться на любой объект, реализующий ее тип интерфейса. При вызове метода для объекта по интерфейсной ссылке выполняется вариант этого метода, реализованный в классе данного объекта. Этот процесс аналогичен применению ссылки на суперкласс для доступа к объекту подкласса (см. главу 7).

Ниже приведен пример программы, демонстрирующий применение интерфейсной ссылки. По такой ссылке в данной программе будут вызываться методы, принадлежащие классам `ByTwos` и `ByThrees`.

```
// Использование интерфейсных ссылок
class ByTwos implements Series {
    int start;
    int val;

    ByTwos() {
        start = 0;
        val = 0;
    }

    public int getNext() {
        val += 2;
        return val;
    }

    public void reset() {
        start = 0;
        val = 0;
    }

    public void setStart(int x) {
        start = x;
        val = x;
    }
}
```

```
class ByThrees implements Series {
   int start;
   int val;

   ByThrees() {
      start = 0;
      val = 0;
   }

   public int getNext() {
      val += 3;
      return val;
   }

   public void reset() {
      start = 0;
      val = 0;
   }

   public void setStart(int x) {
      start = x;
      val = x;
   }
}

class SeriesDemo2 {
   public static void main(String args[]) {
      ByTwos twoOb = new ByTwos();
      ByThrees threeOb = new ByThrees();
      Series ob;

      for(int i=0; i < 5; i++) {
         ob = twoOb;
         System.out.println("Следующее значение ByTwos: " +
                            ob.getNext());
         ob = threeOb;
         System.out.println("Следующее значение Next ByThrees: " +
                            ob.getNext());
      }
   }
}
```

Доступ к объекту с помощью интерфейсной ссылки

В методе `main()` переменная `ob` объявляется как ссылка на интерфейс `Series`. Это означает, что в данной переменной может храниться ссылка на любой объект, реализующий интерфейс `Series`. В данном случае в переменной `ob` сохраняется ссылка на объекты `twoOb` и `threeOb`, т.е. в разные моменты времени переменная представляет собой ссылку на объект класса `ByTwos` или же на объект класса `ByThrees`. Оба этих класса реализуют интерфейс `Series`. Переменная ссылки на интерфейс содержит сведения только о методах, объявленных в этом интерфейсе. Следовательно, переменную `ob` нельзя использовать для доступа к любым другим переменным и методам, которые поддерживаются в объекте, но не объявлены в интерфейсе.

Упражнение 8.1

Создание интерфейса для очереди

ICharQ.java
IQDemo.javajava

Для того чтобы продемонстрировать истинные возможности интерфейсов, обратимся к конкретному практическому примеру. В предыдущих главах был создан класс `Queue`, реализующий простую очередь фиксированного размера для хранения символов. Но обеспечить функционирование очереди можно разными способами. В частности, очередь может быть фиксированного размера или “растущей”, линейной (т.е. переполняться по достижении верхней границы выделенной памяти) или кольцевой (в таком случае при удалении символов из очереди освобождается место для новых элементов). Кроме того, очередь может быть реализована на базе массива, связного списка, двоичного дерева и т.п. Как бы ни была реализована очередь, интерфейс для нее остается неизмененным, т.е. методы `put()` и `get()`, определяющие этот интерфейс, выполняют одинаковые действия независимо от внутренней организации очереди. Поскольку интерфейс для очереди не зависит от ее конкретной реализации, его нетрудно определить, а конкретные детали разработать в каждой реализации очереди по отдельности.

В этом упражнении нам предстоит сначала создать интерфейс для очереди, хранящей символы, а затем реализовать его тремя способами. Во всех трех реализациях для хранения символов будет использоваться массив. Одна из очередей будет линейной и фиксированного размера, т.е. такая же, как и реализованная ранее. Вторая очередь будет кольцевой. В кольцевой очереди по достижении границ массива значения индексов будут автоматически изменяться таким образом, чтобы указывать на начало очереди. Таким образом, в кольцевую очередь можно будет поместить любое количество элементов, но при условии своевременного удаления элементов, включенных в нее ранее. И наконец, третья очередь будет динамической, т.е. ее размеры будут увеличиваться по мере необходимости. Поэтапный процесс создания программы описан ниже.

1. Создайте файл `ICharQ.java` и поместите в него следующее определение интерфейса.

```
// Интерфейс для очереди символов
public interface ICharQ {
   // Помещение символа в очередь
   void put(char ch);

   // Извлечение символа из очереди
   char get();
}
```

   Как видите, этот интерфейс чрезвычайно прост: в нем объявлены только два метода, которые должны быть определены в любом классе, реализующем интерфейс `ICharQ`.

2. Создайте файл `IQDemo.java`.

3. **Начните создавать программу в файле `IQDemo.java`, добавив в него следующий код класса `FixedQueue`.**

```
// Класс, реализующий очередь фиксированного размера
// для хранения символов
class FixedQueue implements ICharQ {
   private char q[];  // массив для хранения элементов очереди
   private int putloc, getloc; // индексы вставляемых и
                               // извлекаемых элементов

   // Создание пустой очереди заданного размера
   public FixedQueue(int size) {
      q = new char[size]; // выделение памяти для очереди
      putloc = getloc = 0;
   }

   // Помещение символа в очередь
   public void put(char ch) {
      if(putloc==q.length) {
         System.out.println(" - Очередь заполнена");
         return;
      }

      q[putloc++] = ch;
   }

   // Извлечение символа из очереди
   public char get() {
      if(getloc == putloc) {
         System.out.println(" - Очередь пуста");
         return (char) 0;
      }

      return q[getloc++];
   }
}
```

Эта реализация интерфейса `ICharQ` выполнена на основе уже знакомого вам класса `Queue`, разработанного в главе 5.

4. **Добавьте в файл `IQDemo.java` приведенный ниже класс `CircularQueue`. Он реализует кольцевую очередь, предназначенную для хранения символов.**

```
// Кольцевая очередь
class CircularQueue implements ICharQ {
   private char q[]; // массив для хранения элементов очереди
   private int putloc, getloc; // индексы вставляемых и
                               // извлекаемых элементов

   // Создание пустой очереди заданного размера
   public CircularQueue(int size) {
      q = new char[size+1]; // выделение памяти для очереди
      putloc = getloc = 0;
   }
```

```
    // Помещение символа в очередь
    public void put(char ch) {
        // Очередь считается полной, если индекс putloc на единицу
        // меньше индекса getloc или если индекс putloc указывает
        // на конец массива, а индекс getloc - на его начало
        if(putloc+1==getloc |
            ((putloc==q.length-1) & (getloc==0))) {
            System.out.println(" - Очередь заполнена");
            return;
        }

        q[putloc++] = ch;
        if(putloc==q.length) putloc = 0; // перейти в начало массива
    }

    // Извлечение символа из очереди
    public char get() {
        if(getloc == putloc) {
            System.out.println(" - Очередь пуста");
            return (char) 0;
        }

        char ch = q[getloc++];
        if(getloc==q.length) getloc = 0; // вернуться в
                                         // начало очереди
        return q[getloc];
    }
}
```

В кольцевой очереди повторно используются элементы массива, освобожденные при извлечении символов. Поэтому в нее можно помещать неограниченное число элементов (при условии, что элементы, помещенные в очередь ранее, будут вовремя удалены). Отслеживание границ массива выполняется очень просто (достаточно обнулить индекс по достижении верхней границы), хотя условие достижения этих границ может на первый взгляд показаться не совсем понятным. Кольцевая очередь переполняется не тогда, когда достигается верхняя граница массива, а тогда, когда число элементов, ожидающих извлечения из очереди, становится слишком большим. Поэтому в методе `put()` проверяется ряд условий с целью определить момент переполнения очереди. Как следует из комментариев к коду, очередь считается заполненной, если индекс `putloc` будет на единицу меньше индекса `getloc` или если индекс `putloc` указывает на конец массива, а индекс `getloc` — на его начало. Как и прежде, очередь считается пустой, если индексы `getloc` и `putloc` равны. С целью упрощения соответствующих проверок размер создаваемого массива на единицу превышает размер очереди.

5. Введите в файл `IQDemo.java` приведенный ниже код класса `DynQueue`. Этот код реализует динамическую, или “растущую”, очередь, т.е. такую, размеры которой увеличиваются, когда в ней не хватает места для символов.

```
// Динамическая очередь
class DynQueue implements ICharQ {
   private char q[];   // массив для хранения элементов очереди
   private int putloc, getloc; // индексы вставляемых и
                               // извлекаемых элементов

   // Создание пустой очереди заданного размера
   public DynQueue(int size) {
      q = new char[size]; // выделение памяти для очереди
      putloc = getloc = 0;
   }

   // Помещение символа в очередь
   public void put(char ch) {
      if(putloc==q.length) {
         // Увеличение размера очереди
         char t[] = new char[q.length * 2];

         // Копирование элементов в новую очередь
         for(int i=0; i < q.length; i++)
            t[i] = q[i];

         q = t;
      }

      q[putloc++] = ch;
   }

   // Извлечение символа из очереди
   public char get() {
      if(getloc == putloc) {
         System.out.println(" - Очередь пуста");
         return (char) 0;
      }

      return q[getloc++];
   }
}
```

В данной реализации при попытке поместить в заполненную очередь еще один элемент создается новый массив, размеры которого в два раза превышают размеры исходного, текущее содержимое очереди копируется в новый массив, а ссылка на него помещается в переменную `q`.

6. Для того чтобы продемонстрировать все три реализации интерфейса `ICharQ`, добавьте в файл `IQDemo.java` приведенный ниже класс, в котором для доступа ко всем трем очередям используется переменная, хранящая ссылку на интерфейс `ICharQ`.

```
// Демонстрация трех реализаций интерфейса ICharQ
class IQDemo {
   public static void main(String args[]) {
```

```
FixedQueue q1 = new FixedQueue(10);
DynQueue q2 = new DynQueue(5);
CircularQueue q3 = new CircularQueue(10);

ICharQ iQ;

char ch;
int i;

iQ = q1;
// Помещение ряда символов в очередь фиксированного размера
for(i=0; i < 10; i++)
   iQ.put((char) ('A' + i));

// Отображение содержимого очереди
System.out.print("Содержимое фиксированной очереди: ");
for(i=0; i < 10; i++) {
   ch = iQ.get();
   System.out.print(ch);
}
System.out.println();

iQ = q2;
// Помещение ряда символов в динамическую очередь
for(i=0; i < 10; i++)
   iQ.put((char) ('Z' - i));

// Отображение содержимого очереди
System.out.print("Содержимое динамической очереди: ");
for(i=0; i < 10; i++) {
   ch = iQ.get();
   System.out.print(ch);
}

System.out.println();

iQ = q3;
// Помещение ряда символов в кольцевую очередь
for(i=0; i < 10; i++)
   iQ.put((char) ('A' + i));

// Отображение содержимого очереди
System.out.print("Содержимое кольцевой очереди: ");
for(i=0; i < 10; i++) {
   ch = iQ.get();
   System.out.print(ch);
}

System.out.println();
```

```
        // Помещение дополнительных символов в кольцевую очередь
        for(i=10; i < 20; i++)
            iQ.put((char) ('A' + i));

        // Отображение содержимого очереди
        System.out.print("Содержимое кольцевой очереди: ");
        for(i=0; i < 10; i++) {
            ch = iQ.get();
            System.out.print(ch);
        }

        System.out.println("\nСохранение и использование данных" +
                           " кольцевой очереди.");

        // Помещение символов в кольцевую очередь с последующим
        // их извлечением
        for(i=0; i < 20; i++) {
            iQ.put((char) ('A' + i));
            ch = iQ.get();
            System.out.print(ch);
        }
    }
}
```

7. Выполнение этой программы дает следующий результат.

```
Содержимое фиксированной очереди: ABCDEFGHIJ
Содержимое динамической очереди: ZYXWVUTSRQ
Содержимое кольцевой очереди: ABCDEFGHIJ
Содержимое кольцевой очереди: KLMNOPQRST
Сохранение и использование данных кольцевой очереди.
ABCDEFGHIJKLMNOPQRST
```

8. А теперь попробуйте поупражняться в организации очередей. Создайте кольцевой вариант очереди `DynQueue`. Добавьте в интерфейс `ICharQ` метод `reset()`, сбрасывающий очередь в исходное состояние. Создайте статический метод для копирования содержимого одной очереди в другую.

Переменные в интерфейсах

Как уже упоминалось выше, в интерфейсах могут объявляться переменные, но они неявно имеют модификаторы `public`, `static` и `final`. На первый взгляд, такие переменные находят лишь ограниченное применение, однако это не совсем так. В крупных программах часто используются константы, описывающие размеры массивов, граничные и специальные значения и т.п. Для крупных программ обычно создается несколько исходных файлов, а следовательно, требуется удобный способ доступа к константам из любого файла. В Java решить эту задачу помогают интерфейсы.

Для того чтобы определить набор общедоступных констант, достаточно создать интерфейс, в котором объявлялись бы не методы, а только нужные константы. Каждый класс, которому требуются эти константы, должен просто реализовать интерфейс, чтобы сделать константы доступными. Ниже приведен простой пример, демонстрирующий данный подход.

```
// Интерфейс, содержащий только константы
interface IConst {
   int MIN = 0;
   int MAX = 10;                                  ── Это константы
   String ERRORMSG = "Ошибка диапазона";
}

class IConstD implements IConst {
   public static void main(String args[]) {
      int nums[] = new int[MAX];

      for(int i=MIN; i < 11; i++) {
         if(i >= MAX) System.out.println(ERRORMSG);
         else {
            nums[i] = i;
            System.out.print(nums[i] + " ");
         }
      }
   }
}
```

Примечание

Относительно целесообразности использования интерфейсов для определения констант ведутся дискуссии. Этот прием описан здесь исключительно ради полноты рассмотрения.

Наследование интерфейсов

Один интерфейс может наследовать другой, для чего служит ключевое слово extends. Синтаксис наследования интерфейсов ничем не отличается от того, который используется для наследования классов. Если класс реализует один интерфейс, наследующий другой, то в нем следует определить все методы, объявленные в интерфейсах по всей цепочке наследования. Ниже приведен пример наследования интерфейсов.

```
// Наследование интерфейсов
interface A {
   void meth1();
   void meth2();
}
```

```
// Интерфейс B содержит методы meth1() и meth2(),
// а кроме того, в него добавляется метод meth3()
interface B extends A { ◄──── Интерфейс B наследует интерфейс A
   void meth3();
}

// Этот класс должен реализовать все методы,
// объявленные в интерфейсах A и B
class MyClass implements B {
   public void meth1() {
      System.out.println("Реализация метода meth1().");
   }

   public void meth2() {
      System.out.println("Реализация метода meth2().");
   }

   public void meth3() {
      System.out.println("Реализация метода meth3().");
   }
}

class IFExtend {
   public static void main(String arg[]) {
      MyClass ob = new MyClass();

      ob.meth1();
      ob.meth2();
      ob.meth3();
   }
}
```

В качестве эксперимента удалите из класса `MyClass` реализацию метода `meth1()`. Это приведет к ошибке при компиляции. Как отмечалось выше, в каждом классе, реализующем интерфейс, должны быть определены все методы, объявленные в интерфейсе, в том числе те, которые были унаследованы от других интерфейсов.

Методы интерфейсов, используемые по умолчанию

Как уже отмечалось, до появления JDK 8 интерфейс не мог определять никакую реализацию вообще. Это означает, что во всех предыдущих версиях Java методы, специфицируемые интерфейсом, могли быть только абстрактными, т.е. не имели тела. Все наши предыдущие обсуждения относятся к интерфейсам именно такой традиционной формы. Выпуск JDK 8 расширил функциональность интерфейсов, добавив возможность определения в них *методов, используемых по умолчанию.* Благодаря этому интерфейс теперь может включать реализации методов, которые будут использоваться по умолчанию, если другие реализации не

предоставляются. Иными словами, теперь метод интерфейса может иметь тело и не быть исключительно абстрактным. В процессе разработки этой идеи для данного средства использовалось также другое название — *метод расширения*, так что вам, скорее всего, будут встречаться оба этих термина.

Основным мотивом для введения методов по умолчанию было стремление обеспечить возможность расширения интерфейсов без разрушения существующего кода (нарушения его работоспособности). Вспомните, что для каждого из методов, определенных в интерфейсе, должна в обязательном порядке предоставляться реализация. В прошлом, если в популярный, широко используемый интерфейс добавлялся новый метод, это приводило к разрушению существующего кода ввиду отсутствия в нем соответствующей реализации. Введение механизма методов по умолчанию разрешает эту проблему, позволяя предоставлять реализацию метода, которая будет использована в том случае, если никакая иная реализация не была предоставлена явно. Таким образом, возможность использования методов по умолчанию обеспечивает сохранение работоспособности существующего кода, даже если интерфейс был обновлен.

Среди других причин, обусловивших появление методов по умолчанию, было желание иметь возможность определять методы интерфейса, являющиеся, по сути, необязательными и используемые лишь при определенных условиях. Например, интерфейс может определять группу методов, воздействующих на последовательности элементов. Один из этих методов может называться `remove()` и предназначаться для удаления элементов последовательности. В то же время, если интерфейс предназначен для поддержки не только изменяемых, но и фиксированных последовательностей, то метод `remove()` является, по сути, опциональным, поскольку в случае фиксированных последовательностей необходимости в его использовании не возникает. До этого класс, реализующий фиксированную последовательность элементов, должен был определять пустую реализацию метода `remove()`, даже если этот метод и не требуется. Теперь же для метода `remove()` в интерфейсе может быть определена реализация по умолчанию, которая либо вообще ничего не делает, либо выводит сообщение об ошибке. Предоставление такой реализации позволяет избавиться от необходимости определять собственную “заглушку” для метода `remove()` в классе, используемом для фиксированных последовательностей. Благодаря этому реализация метода `remove()` классом становится необязательной.

Важно отметить, что появление методов по умолчанию никоим образом не влияет на одно из фундаментальных свойств интерфейсов — интерфейсу по-прежнему запрещено иметь переменные экземпляра. Таким образом, ключевое различие между интерфейсом и классом состоит в том, что класс может поддерживать информацию о состоянии, а интерфейс — не может. Кроме того, создание собственных экземпляров интерфейса, как и ранее, невозможно. Интерфейсы должны реализовываться классами. Поэтому, даже несмотря на то что с выходом JDK 8 у интерфейсов появилась возможность определять методы,

используемые по умолчанию, экземпляры можно создавать только посредством реализации интерфейсов классами.

И последнее замечание. Методы по умолчанию представляют собой специальное средство, обеспечивающее дополнительные возможности. Создаваемые вами интерфейсы будут использоваться главным образом для того, чтобы указать, что именно должно делаться, а не предоставлять способ реализации этих намерений. Вместе с тем включение в интерфейс методов, используемых по умолчанию, обеспечивает дополнительную гибкость в процессе разработки.

Основные сведения о методах по умолчанию

Определение метода по умолчанию в интерфейсе следует той же схеме, что и обычное определение метода в классе. Отличие состоит лишь в том, что объявление метода начинается с ключевого слова `default`. Рассмотрим, например, следующий простой интерфейс.

```
public interface MyIF {
   // Объявление обычного метода интерфейса, которое НЕ включает
   // определение реализации по умолчанию
   int getUserID();

   // Объявление метода по умолчанию, включающее его реализацию
   default int getAdminID() {
      return 1;
   }
}
```

В интерфейсе `MyIF` объявляются два метода. Для первого из них, `getUserID()`, используется стандартное объявление метода интерфейса, не содержащее определение какой-либо реализации. Объявление второго метода, `getAdminID`, включает определение реализации по умолчанию. В данном случае она всего лишь возвращает целочисленное значение 1. Обратите внимание на наличие спецификации `default` в начале объявления. Это общее правило: для определения метода по умолчанию следует предварить его объявление ключевым словом `default`.

Поскольку определение метода `getAdminID()` включает определение реализации по умолчанию, класс, реализующий интерфейс, не обязан переопределять данный метод. Иными словами, если реализующий класс не предоставит собственную реализацию метода, будет использоваться реализация, определенная по умолчанию. Например, приведенное ниже определение класса `MyIFImp` вполне допустимо.

```
// Реализация интерфейса MyIF
class MyIFImp implements MyIF {
   // Реализации подлежит лишь метод getUserID() интерфейса MyIF.
   // Делать это для метода getAdminID() необязательно, поскольку
   // при необходимости может быть использована его реализация,
   // заданная по умолчанию.
```

```
   public int getUserID() {
      return 100;
   }
}
```

Следующий код создает экземпляр класса MyIFImp и использует его для вызова обоих методов, getUserID() и getAdminID().

```
// Использование интерфейсного метода по умолчанию
class DefaultMethodDemo {
   public static void main(String args[]) {

      MyIFImp obj = new MyIFImp();

      // Вызов метода getUserID() возможен, поскольку он явно
      // реализован классом MyIFImp
      System.out.println("Идентификатор пользователя " +
                         obj.getUserID());

      // Вызов метода getAdminID() также возможен, поскольку
      // предоставляется его реализация по умолчанию
      System.out.println("Идентификатор администратора: " +
                         obj.getAdminID());
   }
}
```

В результате выполнения программы будет получен следующий результат.

```
Идентификатор пользователя: 100
Идентификатор администратора: 1
```

Как видите, программа автоматически использовала реализацию метода getAdminID(), заданную по умолчанию. Классу MyIFImp необязательно было реализовывать данный метод. Таким образом, реализация метода getAdminID() является опциональной. (Разумеется, если класс должен возвращать другое значение идентификатора, то собственная реализация метода станет необходимой.)

Класс вполне может — и эта практика является общеупотребительной — определить собственную реализацию метода, определенного по умолчанию в интерфейсе. В качестве примера рассмотрим приведенный ниже класс MyIFImp2, переопределяющий метод getAdminID().

```
class MyIFImp2 implements MyIF {
   // Предоставляются реализации обоих методов -
   // getUserID() и getAdminID()
   public int getUserID() {
      return 100;
   }
   public int getAdminID() {
      return 42;
   }
}
```

Теперь при вызове метода getAdminID() будет возвращено значение, отличное от заданного по умолчанию.

Практический пример использования метода по умолчанию

Предыдущее рассмотрение интерфейсных методов, определяемых по умолчанию, позволило продемонстрировать особенности их применения, однако не показало, насколько они могут быть полезными, на примерах, более приближенных к практике. С этой целью снова обратимся к примеру интерфейса `Series`, рассмотренного ранее. Предположим, интерфейс завоевал широкую популярность и от него зависит работа многих программ. Также допустим, что по результатам исследования во многих случаях в реализации интерфейса `Series` добавлялся метод, который возвращает массив, содержащий следующие `n` элементов ряда. Учитывая это обстоятельство, вы решаете усовершенствовать интерфейс, включив в него такой метод, которому присваивается имя `getNextArray()` и который объявляется следующим образом:

```
int [] getNextArray(int n)
```

где `n` — количество извлекаемых элементов. До появления методов, используемых по умолчанию, добавление этого метода в интерфейс `Series` нарушило бы работоспособность существующего кода, поскольку оказалось бы, что в имеющихся реализациях определение этого метода отсутствует. В то же время предоставление для нового метода версии, используемой по умолчанию, позволяет избежать проблем. Рассмотрим, как это работает.

Иногда при добавлении в существующий интерфейс метода, используемого по умолчанию, в его реализации предусматривают всего лишь вывод сообщения об ошибке. Такой подход требуется использовать тогда, когда для метода по умолчанию невозможно предоставить реализацию, одинаково пригодную для всех возможных случаев его использования. По сути, код подобных методов может быть произвольным. Но иногда удается определить метод по умолчанию, который будет выполнять полезные функции в любом случае. Именно таким является наш метод `getNextArray()`. Поскольку интерфейс `Series` уже содержит требование, чтобы класс реализовал метод `getNext()`, версия по умолчанию `getNextArray()` может использовать его. Таким образом, можно предложить следующий способ реализации новой версии интерфейса `Series`, которая включает используемый по умолчанию метод `getNextArray()`.

```
// Усовершенствованная версия интерфейса Series, которая включает
// используемый по умолчанию метод getNextArray()
public interface Series {
   int getNext(); // возврат следующего числа в ряду

   // Возврат массива, который содержит n элементов,
   // располагающихся в ряду вслед за текущим элементом
   default int[] getNextArray(int n) {
      int[] vals = new int[n];
```

```
        for(int i=0; i < n; i++) vals[i] = getNext();
        return vals;
    }

    void reset(); // сброс
    void setStart(int x); // установка начального значения
}
```

Обратите внимание на то, как реализован метод getNextArray(). Поскольку метод getNext() являлся частью первоначальной спецификации интерфейса Series, он должен предоставляться любым классом, реализующим данный интерфейс. Следовательно, метод getNextArray() может использовать его для получения следующих n элементов ряда. В результате любой класс, реализующий усовершенствованную версию интерфейса Series, сможет использовать метод getNextArray() как есть, без какой-либо необходимости переопределять его. Поэтому работоспособность кода не будет нарушена. Разумеется, класс всегда может предоставить собственную реализацию метода getNextArray(), если это потребуется.

Как показывает предыдущий пример, есть два главных преимущества использования методов по умолчанию:

- это позволяет обновлять интерфейсы, не нарушая работоспособность существующего кода;
- это дает возможность предоставлять дополнительную функциональность и при этом не требовать реализации "заглушек" классами, которым такая функциональность не нужна.

Множественное наследование

Ранее уже отмечалось, что множественное наследование классов в Java не поддерживается. Теперь, когда в состав интерфейсов могут входить методы по умолчанию, вполне закономерно возникает вопрос: а нельзя ли обойти это ограничение с помощью интерфейсов? На этот вопрос следует дать отрицательный ответ. Вспомните о ключевом различии, существующем между классами и интерфейсами: класс может поддерживать состояние (посредством переменных экземпляра), а интерфейс — не может.

И все же механизм методов по умолчанию открывает некоторые возможности, которые обычно связывают с понятием множественного наследования. Например, класс может реализовать два интерфейса. Если каждый из этих интерфейсов предоставляет методы для использования по умолчанию, то определенные аспекты поведения наследуются сразу от обоих интерфейсов. Таким образом, методы по умолчанию в какой-то мере способны обеспечивать поддержку множественного наследования поведения. Нетрудно догадаться, что в подобных ситуациях возможно возникновение конфликтов имен.

Предположим, например, что класс MyClass реализует два интерфейса: Alpha и Beta. Что произойдет в том случае, если оба интерфейса определяют

метод `reset()` и оба предоставляют его реализацию по умолчанию? Какую версию метода будет использовать класс `MyClass`: `Alpha` или `Beta`? А что если класс `MyClass` предоставит собственную реализацию данного метода? Для ответа на эти и другие вопросы в Java определен набор правил, позволяющих разрешать подобные конфликты.

Во-первых, реализация, определенная в классе, всегда имеет более высокий приоритет по сравнению с методами по умолчанию, определенными в интерфейсах. Таким образом, если `MyClass` переопределяет метод по умолчанию `reset()`, добавляя собственную версию данного метода, то использоваться будет именно эта версия. Сказанное относится и к тем ситуациям, в которых класс `MyClass` реализует как интерфейс `Alpha`, так и `Beta`. В этом случае реализация метода, предлагаемая классом `MyClass`, переопределяет обе реализации, заданные по умолчанию.

Во-вторых, в тех случаях, когда класс наследует два интерфейса, определяющих метод по умолчанию с одним и тем же именем, но не переопределяет этот метод, возникает ошибка. В рассматриваемом примере ошибка возникнет в том случае, если класс `MyClass` наследует интерфейсы `Alpha` и `Beta`, но не переопределяет метод `reset()`.

Если же один интерфейс наследует другой, причем оба они определяют метод по умолчанию с одним и тем же именем, то приоритет имеет версия метода, определенная в наследующем интерфейсе. Поэтому, в продолжение рассмотрения нашего примера, если интерфейс `Beta` расширяет интерфейс `Alpha`, то использоваться будет версия метода `reset()`, определенная в интерфейсе `Beta`.

На заданную по умолчанию реализацию можно ссылаться явно, используя ключевое слово `super`. Общая форма подобного обращения приведена ниже:

```
имя_интерфейса.super.имя_метода()
```

Так, если из интерфейса `Beta` необходимо обратиться к методу по умолчанию интерфейса `Alpha`, то для этого можно воспользоваться следующей инструкцией:

```
Alpha.super.reset();
```

Использование статических методов интерфейса

В JDK 8 была добавлена еще одна возможность: теперь интерфейс может определять один или несколько статических методов. Как и статические методы класса, статические методы, определенные в интерфейсе, можно вызывать без привлечения объектов. Таким образом, чтобы вызвать статический метод, ни реализация интерфейса, ни создание его экземпляра не требуются. Для вызова статического метода достаточно указать имя его интерфейса, а после него, используя точечную нотацию, имя самого метода. Ниже приведена общая форма такого вызова:

```
имя_интерфейса.имя_статического_метода();
```

Обратите внимание на сходство этой формы с формой вызова статического метода класса.

Ниже приведен пример добавления статического метода `getUniversalID()`, возвращающего 0, в рассмотренный ранее интерфейс `MyIF`.

```
public interface MyIF {
   // Объявление обычного метода интерфейса, которое НЕ включает
   // определение реализации по умолчанию
   int getUserID();

   // Объявление метода по умолчанию, включающее его реализацию
   default int getAdminID() {
      return 1;
   }

   // Объявление статического метода интерфейса
   static int getUniversalID() {
      return 0;
   }
}
```

А вот пример вызова метода `getUniversalID()`:

```
int uID = MyIF.getUniversalID();
```

Как уже упоминалось, никакой реализации интерфейса `MyIF` или создания его экземпляра для вызова метода `getUniversalID()` не требуется, поскольку он статический.

Следует также отметить, что статические методы интерфейса не наследуются ни реализующим его классом, ни производными интерфейсами.

Закрытые методы интерфейса

Начиная с JDK 9 интерфейс может включать закрытые методы. Закрытый метод интерфейса может вызваться только методом, заданным по умолчанию, или другим закрытым методом, определенным в том же самом интерфейсе. И поскольку закрытый метод интерфейса определен с помощью ключевого слова `private`, его невозможно использовать в коде, находящемся за пределами интерфейса, в котором этот метод определен. Это ограничение включает подчиненные интерфейсы, поскольку закрытый метод интерфейса не наследуется ими.

Основное преимущество закрытого метода интерфейса заключается в том, что он позволяет двум или более методам, заданным по умолчанию, использовать общий фрагмент кода, позволяя избежать дублирования кода. Например, рассмотрим расширенную версию интерфейса `Series`, в которую включен второй метод, заданный по умолчанию, `skipAndGetNextArray()`. Этот метод пропускает заданное число элементов, а затем возвращает массив, который содержит последующие элементы. При этом используется закрытый метод `getArray()` для получения массива элементов заданного размера.

```
// Еще одна расширенная версия интерфейса Series, включающая
// два заданных по умолчанию метода и использующая закрытый
// метод getArray();
public interface Series {
   int getNext(); // возвращает следующее число в ряду

   // Возврат массива, который содержит следующие n элементов
   // ряда, помимо текущего элемента
   default int[] getNextArray(int n) {
      return getArray(n);
   }

   // Возврат массива, содержащего следующие n элементов
   // в ряду, после пропуска элементов
   default int[] skipAndGetNextArray(int skip, int n) {
      // Пропуск указанного числа элементов
      getArray(skip);

      return getArray(n);
   }

   // Закрытый метод, возвращающий массив, который
   // содержит следующие n элементов
   private int[] getArray(int n) {
      int[] vals = new int[n];

      for(int i=0; i < n; i++) vals[i] = getNext();
      return vals;
   }

   void reset(); // перезапуск
   void setStart(int x); // установка начального значения
}
```

Обратите внимание на то, что оба метода, `getNextArray()` и `skipAndGetNextArray()`, используют закрытый метод `getArray()` для получения возвращаемого массива. Благодаря этому предотвращается дублирование кода этими методами. Учтите, что, поскольку метод `getArray()` является закрытым, его нельзя вызвать в коде, находящемся за пределами интерфейса `Series`. Использование этого метода ограничено методами по умолчанию, входящими в интерфейс `Series`.

Несмотря на то что закрытые методы интерфейса нужны не так часто, они все же являются достаточно полезными.

Итоговые замечания относительно пакетов и интерфейсов

В примерах книги пакеты и интерфейсы используются редко, однако оба этих средства являются важной частью Java. Практически любая реальная

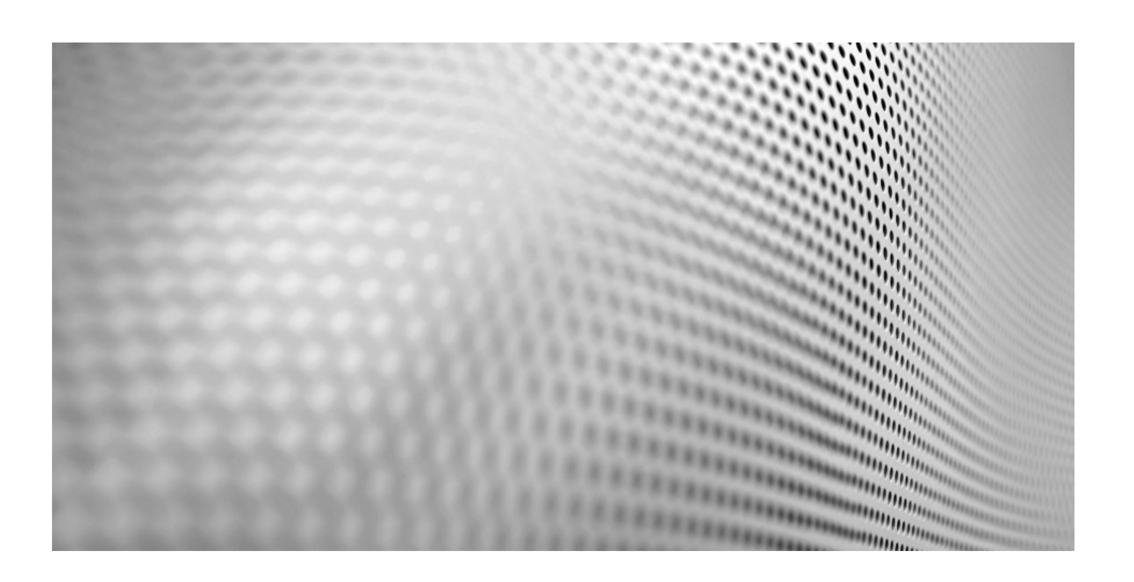
программа, которую вам придется написать на Java, будет находиться в каком-либо пакете. Точно так же можно не сомневаться, что многие из ваших программ будут включать в себя интерфейсы. Как будет продемонстрировано в главе 15, пакеты играют важную роль при работе с модулями, которые появились в JDK 9. Поэтому навыки работы с пакетами и интерфейсами будет очень полезны.

Вопросы и упражнения для самопроверки

1. Используя код, созданный в упражнении 8.1, поместите в пакет `qpack` интерфейс `ICharQ` и все три реализующих его класса. Оставив класс `IQDemo` в пакете, используемом по умолчанию, покажите, как импортировать и использовать классы из пакета `qpack`.
2. Что такое пространство имен? Почему так важна возможность его разделения на отдельные области в Java?
3. Содержимое пакетов хранится в ______________.
4. В чем отличие доступа, определяемого ключевым словом `protected`, от доступа по умолчанию?
5. Допустим, классы, содержащиеся в одном пакете, требуется использовать в другом пакете. Какими двумя способами можно этого добиться?
6. "Один интерфейс — множество методов" — таков главный принцип Java. Какое языковое средство лучше всего демонстрирует этот принцип?
7. Сколько классов могут реализовать один и тот же интерфейс? Сколько интерфейсов может реализовать класс?
8. Может ли один интерфейс наследовать другой?
9. Создайте интерфейс для класса `Vehicle`, рассмотренного в главе 7, назвав его `IVehicle`.
10. Переменные, объявленные в интерфейсе, неявно имеют модификаторы `static` и `final`. Какие преимущества это дает?
11. Пакет по сути является контейнером для классов. Верно или не верно?
12. Какой стандартный пакет автоматически импортируется в любую программу на Java?
13. Какое ключевое слово используется для объявления в интерфейсе метода по умолчанию?
14. Допускается ли, начиная с JDK 8, определение статического метода интерфейса?
15. Предположим, что интерфейс `ICharQ`, представленный в упражнении 8.1, получил широкое распространение в течение нескольких лет. В какой-то

момент вы решили добавить в него метод `reset()`, который будет использоваться для сброса очереди в ее исходное пустое состояние. Как это можно осуществить, не нарушая работоспособность существующего кода, в случае использования комплекта JDK 8 или выше?

16. Как можно вызвать статический метод интерфейса?
17. Может ли интерфейс включать закрытый (`private`) метод?



Глава 9

Обработка исключений

В этой главе...

- Иерархия исключений
- Инструкции `try` и `catch`
- Необработанные исключения
- Множественные инструкции `catch`
- Перехват исключений подклассов
- Вложенные блоки `try`
- Генерирование исключений
- Класс `Throwable`
- Ключевое слово `finally`
- Ключевое слово `throws`
- Встроенные классы исключений Java
- Создание специальных классов исключений

В этой главе речь пойдет об обработке исключительных ситуаций, или, как говорят, исключений. Исключение — это ошибка, возникающая в процессе выполнения программы. Используя подсистему обработки исключений Java, можно управлять реакцией программы на появление ошибок во время выполнения. Средства обработки исключений в том или ином виде имеются практически во всех современных языках программирования. Можно смело утверждать, что в Java подобные инструментальные средства отличаются большей гибкостью, понятнее и удобнее в применении по сравнению с большинством других языков программирования.

Преимущество обработки исключений заключается в том, что она предусматривает автоматическую реакцию на многие ошибки, избавляя от необходимости писать вручную соответствующий код. Например, в некоторых устаревших языках программирования предусматривается возврат специального кода при возникновении ошибки в ходе выполнения метода. Этот код приходится проверять вручную при каждом вызове метода. Подобный подход к обработке ошибок вручную трудоемок и чреват сбоями. Обработка исключений упрощает этот процесс, предоставляя возможность определять в программе блок кода, называемый *обработчиком исключения* и автоматически выполняющийся при возникновении ошибки. Это избавляет от необходимости проверять вручную, насколько удачно или неудачно была выполнена та или иная операция. Если

возникнет ошибка, все необходимые действия по ее устранению выполнит обработчик исключений.

Для наиболее часто встречающихся программных ошибок, в том числе деления на нуль или попытки открыть несуществующий файл, в Java определены стандартные исключения. Чтобы обеспечить требуемую реакцию на конкретную ошибку, в программу следует включить соответствующий обработчик событий. Исключения широко применяются в стандартной библиотеке Java API.

Другими словами, для успешного программирования на Java нужно уметь обращаться с подсистемой обработки исключений, предусмотренной в этом языке программирования.

Иерархия исключений

В Java все исключения представлены отдельными классами. Все классы исключений являются потомками класса `Throwable`. Так, если в программе возникнет исключительная ситуация, будет сгенерирован объект класса, соответствующего определенному типу исключения. У класса `Throwable` имеются два непосредственных подкласса: `Exception` и `Error`. Исключения типа `Error` относятся к ошибкам, возникающим в виртуальной машине Java, а не в прикладной программе. Контролировать такие исключения невозможно, поэтому реакция на них в приложении, как правило, не предусматривается. В связи с этим исключения данного типа не будут рассматриваться в книге.

Ошибки, связанные с работой программы, представлены отдельными подклассами, производными от класса `Exception`. В частности, к этой категории относятся ошибки деления на нуль, выхода за пределы массива и обращения к файлам. Подобные ошибки следует обрабатывать в самой программе. Важным подклассом, производным от `Exception`, является класс `RuntimeException`, который служит для представления различных видов ошибок, часто возникающих во время выполнения программ.

Общие сведения об обработке исключений

Для обработки исключений в Java предусмотрены пять ключевых слов: `try`, `catch`, `throw`, `throws` и `finally`. Они образуют единую подсистему, в которой использование одного ключевого слова почти всегда автоматически влечет за собой употребление другого. Каждое из упомянутых выше ключевых слов будет подробно рассмотрено далее. Но прежде следует получить общее представление об их роли в процессе обработки исключений. Поэтому ниже вкратце поясняется, каким образом они действуют.

Инструкции, в которых требуется отслеживать появление исключений, заключаются в блок `try`. Если в блоке `try` будет сгенерировано исключение, его можно перехватить и обработать нужным образом. Системные исключения генерируются автоматически. А для того чтобы сгенерировать исключение

вручную, следует воспользоваться инструкцией throw. Иногда возникает потребность обрабатывать исключения за пределами метода, в котором они возникают, и в этом случае необходимо указывать их с помощью ключевого слова throws. Код, который в любом случае должен быть выполнен после выхода из блока try, помещается в блок finally.

Спросим у эксперта

ВОПРОС. Можно ли подробнее описать условия, при которых генерируются исключения?

ОТВЕТ. Исключения генерируются при выполнении одного из трех условий. Во-первых, исключение может сгенерировать виртуальная машина Java в качестве реакции на некоторую внутреннюю ошибку, но такие исключения в прикладных программах не контролируются. Во-вторых, причиной исключения может стать ошибка в коде программы. Примерами подобных ошибок являются деление на нуль и выход за пределы массива. Такие исключения подлежат обработке в прикладных программах. И в-третьих, исключения можно сгенерировать вручную, используя ключевое слово throw. Порядок обработки исключений не зависит от того, как именно они были сгенерированы.

Использование инструкций try и catch

Основными языковыми средствами обработки исключений являются инструкции try и catch. Они используются совместно. Это означает, что в коде нельзя указать ключевое слово catch, не указав ключевого слова try. Ниже приведена общая форма записи блоков try/catch, предназначенных для обработки исключений.

```
try {
   // блок кода, в котором должны отслеживаться ошибки
}

catch (тип_исключения_1 объект_исключения) {
   // обработчик исключения тип_исключения_1
}

catch (тип_исключения_2 объект_исключения) {
   // обработчик исключения тип_исключения_2
}
.
.
.
```

В скобках, следующих за ключевым словом catch, указываются тип исключения и переменная, ссылающаяся на объект данного типа. Когда возникает исключение, оно перехватывается соответствующей инструкцией catch,

обрабатывающей это исключение. Как следует из приведенной выше общей формы записи, с одним блоком `try` может быть связано несколько инструкций `catch`. Тип исключения определяет, какая именно инструкция `catch` будет выполняться. Так, если тип исключения соответствует спецификации инструкции `catch`, то именно она и будет выполнена, а остальные инструкции `catch` — пропущены. При перехвате исключения переменной, указанной в скобках после ключевого слова `catch`, присваивается ссылка на *объект_исключения*.

Следует иметь в виду, что если исключение не генерируется, то блок `try` завершается обычным образом, и ни одна из его инструкций `catch` не выполняется. Выполнение программы продолжается с первой инструкции, следующей за последней инструкцией `catch`. Таким образом, инструкции `catch` выполняются только при появлении исключения.

Примечание

Существует форма инструкции `try`, поддерживающая *автоматическое управление ресурсами* и называемая инструкцией `try` *с ресурсами*. Более подробно эта форма инструкции `try` будет описана в главе 10 при рассмотрении потоков ввода-вывода, в том числе файловых, поскольку потоки ввода-вывода относятся к числу ресурсов, наиболее часто используемых в приложениях.

Простой пример обработки исключений

Рассмотрим простой пример, демонстрирующий перехват и обработку исключения. Как известно, попытка обратиться за пределы массива приводит к ошибке, и виртуальная машина Java генерирует соответствующее исключение `ArrayIndexOutOfBoundsException`. Ниже приведен код программы, в которой намеренно создаются условия для появления данного исключения, которое затем перехватывается.

```
// Демонстрация обработки исключений
class ExcDemo1 {
   public static void main(String args[]) {
      int nums[] = new int[4];

      try {    <------- Создание блока try
         System.out.println("До генерации исключения");

         nums[7] = 10;  <--------------- Попытка выйти за пределы массива nums
         System.out.println("Эта строка не будет отображаться");
      }
      catch (ArrayIndexOutOfBoundsException exc) {  <---- Перехват ошибок,
         System.out.println("Выход за пределы массива!");   вызываемых выходом
      }                                                      за пределы массива
      System.out.println("После инструкции catch");
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
До генерации исключения
Выход за пределы массива!
После инструкции catch
```

Несмотря на всю простоту данного примера программы он наглядно демонстрирует несколько важных особенностей обработки исключений. Во-первых, контролируемый код помещается в блок try. И во-вторых, когда возникает исключение (в данном случае это происходит при попытке использования индекса, выходящего за пределы массива), выполнение блока try прерывается и управление получает блок catch. Следовательно, явного обращения к блоку catch не происходит, но переход к нему осуществляется лишь при определенном условии, возникающем в ходе выполнения программы. Так, инструкция вызова метода println(), следующая за выражением, в котором происходит обращение к несуществующему элементу массива, вообще не может быть выполнена. При выходе из блока catch выполнение программы продолжается с инструкции, следующей за этим блоком. Таким образом, обработчик исключений предназначен для устранения программных ошибок, приводящих к исключительным ситуациям, а также для обеспечения нормального продолжения выполняемой программы.

Как упоминалось выше, в отсутствие появления исключений в блоке try инструкции в блоке catch управления не получают, и выполнение программы продолжается после блока catch. Для того чтобы убедиться в этом, замените в предыдущей программе строку кода

```
nums[7] = 10;
```

следующей строкой:

```
nums[0] = 10;
```

Теперь исключение не возникнет, и блок catch не выполнится.

Важно понимать, что исключения отслеживаются во всем коде, находящемся в блоке try. Это относится и к исключениям, которые могут быть сгенерированы методом, вызываемым из блока try. Исключения, возникающие в вызываемом методе, перехватываются инструкциями в блоке catch, связанном с этим блоком try. Правда, это произойдет лишь в том случае, если метод не обрабатывает исключения самостоятельно. Рассмотрим в качестве примера следующую программу.

```
/* Исключение может быть сгенерировано одним методом,
   а перехвачено другим */

class ExcTest {
   // Генерация исключения
   static void genException() {
      int nums[] = new int[4];

      System.out.println("До генерации исключения");
```

```
      // Выход за пределы массива
      nums[7] = 10; ◄——— Здесь генерируется исключение
      System.out.println("Эта строка не будет отображаться");
   }
}

class ExcDemo2 {
   public static void main(String args[]) {

      try {
         ExcTest.genException();
      }
      catch (ArrayIndexOutOfBoundsException exc) { ◄——— Здесь перехватывается исключение
         System.out.println("Выход за пределы массива!");
      }
      System.out.println("После инструкции catch");
   }
}
```

Выполнение этой версии программы дает тот же результат, что и предыдущая версия.

```
До генерации исключения
Выход за пределы массива!
После инструкции catch
```

Метод `genException()` вызывается из блока `try`, и поэтому генерируемое, но не перехватываемое в нем исключение перехватывается далее в блоке `catch`, находящемся в методе `main()`. Если бы метод `genException()` сам перехватывал исключение, оно вообще не достигло бы метода `main()`.

Необработанные исключения

Перехват стандартного исключения Java, продемонстрированный в предыдущем примере, позволяет предотвратить завершение программы вследствие ошибки. Генерируемое исключение должно быть перехвачено и обработано. Если исключение не обрабатывается в программе, оно будет обработано виртуальной машиной Java. Но это закончится тем, что по умолчанию виртуальная машина Java аварийно завершит программу, выведя сообщение об ошибке и трассировку стека исключений. Допустим, в предыдущем примере попытка обращения за пределы массива не отслеживается и исключение не перехватывается.

```
// Обработка ошибки средствами виртуальной машины Java
class NotHandled {
   public static void main(String args[]) {
     int nums[] = new int[4];

      System.out.println("До генерации исключения");
```

```
        // Сгенерировать исключение в связи с
        // выходом индекса за пределы массива
        nums[7] = 10;
    }
}
```

При появлении ошибки, связанной с обращением за пределы массива, выполнение программы прекращается, и выводится следующее сообщение.

```
Exception in thread "main"
   java.lang.ArrayIndexOutOfBoundsException: 7
      at NotHandled.main(NotHandled.java:9)
```

Такие сообщения полезны на этапе отладки, но пользователям программы эта информация вряд ли нужна. Именно поэтому очень важно, чтобы программы обрабатывали исключения самостоятельно и не поручали эту задачу виртуальной машине Java.

Как упоминалось выше, тип исключения должен соответствовать типу, указанному в инструкции `catch`. В противном случае исключение не будет перехвачено. Так, в приведенном ниже примере программы делается попытка перехватить исключение, связанное с выходом индекса за пределы массива, с помощью инструкции `catch`, в которой указан тип `ArithmeticException` — еще одно встроенное исключение Java. При некорректном обращении к массиву будет сгенерировано исключение `ArrayIndexOutOfBoundsException`, не соответствующее типу, указанному в инструкции `catch`. В результате программа будет завершена аварийно.

```
// Эта программа не будет работать!
class ExcTypeMismatch {
   public static void main(String args[]) {
      int nums[] = new int[4];

      try {
         System.out.println("До генерации исключения");

         // Сгенерировать исключение в связи с
         // выходом индекса за пределы массива
         nums[7] = 10;  // ◄ Генерирование исключения ArrayIndexOutBoundsException
         System.out.println("Эта строка не будет отображаться");
      }

      // Исключение, связанное с обращением за пределы массива,
      // нельзя обработать с помощью инструкции catch, в которой
      // указан тип исключения ArithmeticException
      catch (ArithmeticException exc) {  // ◄ Попытка перехвата исключения ArithmeticException
         // Перехватить исключение
         System.out.println("Выход за пределы массива!");
      }
      System.out.println("После инструкции catch");
   }
}
```

Ниже приведен результат выполнения данной программы.

```
До генерации исключения
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 7
        at ExcTypeMismatch.main(ExcTypeMismatch.java:10)
```

Нетрудно заметить, что инструкция `catch`, в которой указан тип исключения `ArithmeticException`, не может перехватить исключение `ArrayIndexOutOfBoundsException`.

Обработка исключений — изящный способ устранения программных ошибок

Одно из главных преимуществ обработки исключений заключается в том, что это позволяет вовремя отреагировать на ошибку в программе и затем продолжить ее выполнение. В качестве примера рассмотрим еще одну программу, в которой элементы одного массива делятся на элементы другого. Если при этом происходит деление на нуль, то генерируется исключение `ArithmeticException`. Обработка подобного исключения заключается в том, что программа уведомляет об ошибке и затем продолжает свое выполнение. Таким образом, попытка деления на нуль не приведет к аварийному завершению программы из-за появления ошибки во время выполнения. Вместо этого осуществляется корректная обработка ошибки, не прерывающая выполнения программы.

```
// Корректная обработка исключения и продолжение
// выполнения программы
class ExcDemo3 {
   public static void main(String args[]) {
      int numer[] = { 4, 8, 16, 32, 64, 128 };
      int denom[] = { 2, 0, 4, 4, 0, 8 };

      for(int i=0; i<numer.length; i++) {
         try {
            System.out.println(numer[i] + " / " +
                               denom[i] + " равно " +
                               numer[i]/denom[i]);
         }
         catch (ArithmeticException exc) {
            // Перехват исключения
            System.out.println("Попытка деления на нуль!");
         }
      }
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
4 / 2 равно 2
Попытка деления на нуль!
```

```
16 / 4 равно 4
32 / 4 равно 8
Попытка деления на нуль!
128 / 8 равно 16
```

Данный пример демонстрирует еще одну важную особенность: обработанное исключение удаляется из системы. Иными словами, на каждом шаге цикла блок `try` выполняется заново, а все возникшие ранее исключения считаются обработанными. Благодаря этому в программе могут обрабатываться повторно возникающие ошибки.

Множественные блоки catch

Как пояснялось ранее, с блоком `try` можно связать несколько инструкций `catch`. Обычно разработчики так и поступают на практике. Каждая из инструкций `catch` должна перехватывать отдельный тип исключений. Например, в приведенной ниже программе обрабатываются как исключения, связанные с выходом за пределы массива, так и ошибки деления на нуль.

```
// Применение нескольких инструкций catch
class ExcDemo4 {
   public static void main(String args[]) {
      // Длина массива numer превышает длину массива denom
      int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
      int denom[] = { 2, 0, 4, 4, 0, 8 };

      for(int i=0; i<numer.length; i++) {
         try {
            System.out.println(numer[i] + " / " +
                               denom[i] + " равно " +
                               numer[i]/denom[i]);
         }
         catch (ArithmeticException exc) {  ◄── Несколько инструкций catch
            // Перехват исключения
            System.out.println("Попытка деления на нуль!");
         }
         catch (ArrayIndexOutOfBoundsException exc) {  ◄──┘
            // Перехватить исключение
            System.out.println("Соответствующий элемент не найден");
         }
      }
   }
}
```

Выполнение этой программы дает следующий результат.

```
4 / 2 равно 2
Попытка деления на нуль!
16 / 4 равно 4
32 / 4 равно 8
Попытка деления на нуль!
```

```
128 / 8 равно 16
Соответствующий элемент не найден
Соответствующий элемент не найден
```

Как подтверждает приведенный выше результат выполнения программы, в каждом блоке catch обрабатывается свой тип исключения.

Вообще говоря, выражения с инструкциями catch проверяются в том порядке, в котором они встречаются в программе. И выполняется лишь та из них, которая соответствует типу возникшего исключения. Остальные блоки catch просто игнорируются.

Перехват исключений, генерируемых подклассами

В случае использования множественных инструкций catch важно знать об одной интересной особенности: условие перехвата исключений для суперкласса будет справедливо и для любых его подклассов. Например, класс Throwable является суперклассом для всех исключений, поэтому для перехвата всех возможных исключений в инструкциях catch следует указывать тип Throwable. Если же требуется перехватывать исключения суперкласса и подкласса, то в блоке инструкций первым должен быть указан тип исключения, генерируемого подклассом. В противном случае вместе с исключением суперкласса будут перехвачены и все исключения производных от него классов. Это правило соблюдается автоматически, так что указание сначала исключения суперкласса, а затем всех остальных приведет к созданию недостижимого кода, поскольку условие перехвата исключения, генерируемого подклассом, никогда не будет выполнено. При этом следует учитывать, что в Java наличие недостижимого кода считается ошибкой.

Рассмотрим в качестве примера следующую программу.

```
// В инструкциях catch исключения подкласса должны
// предшествовать исключениям суперкласса
class ExcDemo5 {
   public static void main(String args[]) {
      // Длина массива numer превышает длину массива denom
      int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
      int denom[] = { 2, 0, 4, 4, 0, 8 };

      for(int i=0; i<numer.length; i++) {
         try {
            System.out.println(numer[i] + " / " +
                               denom[i] + " равно " +
                               numer[i]/denom[i]);
         }
         catch (ArrayIndexOutOfBoundsException exc) {  ◄── Перехват подкласса
            // Перехват исключения
            System.out.println("Соответствующий элемент не найден");
         }
```

```
         catch (Throwable exc) { ◄──────────── Перехват суперкласса
            System.out.println("Возникло исключение");
         }
      }
   }
}
```

Ниже приведен результат выполнения данной программы.

```
4 / 2 равно 2
Возникло исключение
16 / 4 равно 4
32 / 4 равно 8
Возникло исключение
128 / 8 равно 16
Соответствующий элемент не найден
Соответствующий элемент не найден
```

В данном случае инструкция `catch(Throwable)` перехватывает все исключения, кроме `ArrayIndexOutOfBoundsException`. Соблюдение правильного порядка следования инструкций `catch` приобретает особое значение в тех случаях, когда исключения генерируются в самой программе.

Спросим у эксперта

ВОПРОС. Зачем перехватывать исключения, генерируемые суперклассами?

ОТВЕТ. На то могут быть самые разные причины. Ограничимся рассмотрением двух из них. Во-первых, включив в программу блок `catch` для перехвата исключений типа `Exception`, вы получаете универсальный механизм обработки всех исключений, связанных с выполнением вашей программы. Такой универсальный обработчик исключений оказывается удобным, например, в тех случаях, когда требуется предотвратить аварийное завершение программы, независимо от возникшей ситуации. И во-вторых, для обработки некоторой категории исключений иногда подходит единый алгоритм. Перехватывая эти исключения, генерируемые суперклассом, можно избежать дублирования кода.

Вложенные блоки `try`

Блоки `try` могут быть вложенными один в другой. Исключение, возникшее во внутреннем блоке `try` и не перехваченное связанным с ним блоком `catch`, распространяется далее во внешний блок `try` и обрабатывается связанным с ним блоком `catch`. Такой порядок обработки исключений демонстрируется в приведенном ниже примере программы, где исключение `ArrayIndexOutOfBoundsException` не перехватывается во внутреннем блоке `catch`, но обрабатывается во внешнем.

```
// Использование вложенных блоков try
class NestTrys {
   public static void main(String args[]) {
      // Длина массива numer превышает длину массива denom
      int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
      int denom[] = { 2, 0, 4, 4, 0, 8 };

      try { // внешний блок try          ◄──────── Вложенные блоки try
         for(int i=0; i<numer.length; i++) {
            try { // внутренний блок try ◄────────
               System.out.println(numer[i] + " / " +
                                  denom[i] + " равно " +
                                  numer[i]/denom[i]);
            }
            catch (ArithmeticException exc) {
               // Перехват исключения
               System.out.println("Попытка деления на нуль");
            }
         }
      }
      catch (ArrayIndexOutOfBoundsException exc) {
         // Перехват исключения
         System.out.println("Соответствующий элемент не найден");
         System.out.println("Фатальная ошибка - выполнение программы
                         прервано!");
      }
   }
}
```

Выполнение этой программы может дать, например, следующий результат.

```
4 / 2 равно 2
Попытка деления на нуль
16 / 4 равно 4
32 / 4 равно 8
Попытка деления на нуль
128 / 8 равно 16
Соответствующий элемент не найден
Фатальная ошибка - выполнение программы прервано!
```

В данном примере исключение, которое может быть обработано во внутреннем блоке `try` (в данном случае ошибка деления на нуль), не мешает дальнейшему выполнению программы. А вот ошибка выхода за пределы массива перехватывается во внешнем блоке `try`, что приводит к аварийному завершению программы.

Ситуация, продемонстрированная в предыдущем примере, является не единственной причиной для применения вложенных блоков `try`, хотя она встречается очень часто. В этом случае вложенные блоки `try` помогают по-разному обрабатывать разные типы ошибок. Одни ошибки невозможно устранить, а для других достаточно предусмотреть сравнительно простые действия. Внешний блок `try` чаще всего используется для перехвата критических ошибок, а менее серьезные ошибки обрабатываются во внутреннем блоке `try`.

Генерирование исключений

В предыдущих примерах программ обрабатывались исключения, автоматически генерируемые виртуальной машиной Java. Но генерировать исключения можно и вручную, используя для этого инструкцию `throw`. Вот общая форма этой инструкции:

```
throw объект_исключения;
```

где *объект_исключения* должен быть объектом класса, производного от класса `Throwable`.

Ниже приведен пример программы, демонстрирующий применение инструкции `throw`. В этой программе исключение `ArithmeticException` генерируется вручную.

```
// Генерирование исключения вручную
class ThrowDemo {
   public static void main(String args[]) {
      try {
         System.out.println("До инструкции throw");
         throw new ArithmeticException();  ◄——— Генерирование исключения
      }
      catch (ArithmeticException exc) {
         // Перехват исключения
         System.out.println("Исключение перехвачено");
      }
      System.out.println("После блока try/catch");
   }
}
```

Выполнение этой программы дает следующий результат.

```
До инструкции throw
Исключение перехвачено
После блока try/catch
```

Обратите внимание на то, что исключение `ArithmeticException` генерируется с помощью ключевого слова `new` в инструкции `throw`. Дело в том, что инструкция `throw` генерирует исключение в виде объекта. Поэтому после ключевого слова `throw` недостаточно указать только тип исключения, нужно еще создать объект для этой цели.

Повторное генерирование исключений

Исключение, перехваченное блоком `catch`, может быть повторно сгенерировано для обработки другим аналогичным блоком. Чаще всего повторное генерирование исключений применяется с целью предоставить разным обработчикам доступ к исключению. Так, например, повторное генерирование исключения имеет смысл в том случае, когда один обработчик оперирует одним свойством исключения, а другой ориентирован на другое его свойство. Повторно сгенерированное исключение не может быть перехвачено тем же самым блоком `catch`. Оно распространяется в следующий блок `catch`.

Спросим у эксперта

ВОПРОС. Зачем генерировать исключения вручную?

ОТВЕТ. Чаще всего генерируемые вручную исключения являются экземплярами создаваемых классов исключений. Как будет показано далее, создание собственных классов исключений позволяет обрабатывать ошибки в рамках единой стратегии обработки исключений в разрабатываемой прикладной программе.

Ниже приведен пример программы, демонстрирующий повторное генерирование исключений.

```
// Повторное генерирование исключений
class Rethrow {
   public static void genException() {
      // Длина массива numer превышает длину массива denom
      int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
      int denom[] = { 2, 0, 4, 4, 0, 8 };

      for(int i=0; i<numer.length; i++) {
         try {
            System.out.println(numer[i] + " / " +
                               denom[i] + " равно " +
                               numer[i]/denom[i]);
         }
         catch (ArithmeticException exc) {
            // Перехват исключения
            System.out.println("Попытка деления на нуль");
         }
         catch (ArrayIndexOutOfBoundsException exc) {
            // Перехват исключения
            System.out.println("Соответствующий элемент не найден");
            throw exc; // повторно сгенерировать исключение
         }          ↑
      }             └─────────── Повторное генерирование исключения
   }
}

class RethrowDemo {
   public static void main(String args[]) {
      try {
         Rethrow.genException();
      }
      catch(ArrayIndexOutOfBoundsException exc) { ◄── Перехват повторно
                                                      сгенерированного исключения
         // Повторный перехват исключения
         System.out.println("Фатальная ошибка - " +
                            "выполнение программы прервано!");
      }
   }
}
```

В данной программе ошибка деления на нуль обрабатывается локально в методе `genException()`, а при попытке обращения за пределы массива исключение генерируется повторно. На этот раз оно перехватывается в методе `main()`.

Подробнее о классе Throwable

В приведенных до сих примерах программ только перехватывались исключения, но не выполнялось никаких действий над представляющими их объектами. В выражении инструкции `catch` указываются тип исключения и параметр, принимающий объект исключения. А поскольку все исключения представлены подклассами, производными от класса `Throwable`, то они поддерживают методы, определенные в этом классе. Некоторые наиболее часто используемые методы из класса `Throwable` приведены в табл. 9.1.

Таблица 9.1. Наиболее часто используемые методы из класса Throwable

Метод	Описание
`Throwable fillInStackTrace()`	Возвращает объект типа `Throwable`, содержащий полную трассировку стека исключений. Этот объект пригоден для повторного генерирования исключений
`String getLocalizedMessage()`	Возвращает описание исключения, локализованное по региональным стандартам
`String getMessage()`	Возвращает описание исключения
`void printStackTrace()`	Выводит трассировку стека исключений
`void printStackTrace (PrintStream` *поток*`)`	Выводит трассировку стека исключений в указанный поток
`void printStackTrace (PrintWriter` *поток*`)`	Направляет трассировку стека исключений в указанный поток
`String toString()`	Возвращает объект типа `String`, содержащий полное описание исключения. Этот метод вызывается из метода `println()` при выводе объекта типа `Throwable`

Среди методов, определенных в классе `Throwable`, наибольший интерес представляют методы `printStackTrace()` и `toString()`. С помощью метода `printStackTrace()` можно вывести стандартное сообщение об ошибке и запись последовательности вызовов методов, которые привели к возникновению исключения. А метод `toString()` позволяет получить стандартное сообщение об ошибке. Он также вызывается в том случае, когда объект исключения передается в качестве параметра методу `println()`. Применение этих методов демонстрируется в следующем примере программы.

```
// Использование методов класса Throwable
class ExcTest {
```

```
   static void genException() {
      int nums[] = new int[4];

      System.out.println("До генерации исключения");

      // Генерирование исключения в связи с
      // выходом индекса за пределы массива
      nums[7] = 10;
      System.out.println("Эта строка не будет отображаться");
   }
}

class UseThrowableMethods {
   public static void main(String args[]) {

      try {
         ExcTest.genException();
      }
      catch (ArrayIndexOutOfBoundsException exc) {
         // перехват исключения
         System.out.println("Стандартное сообщение: ");
         System.out.println(exc);
         System.out.println("\nСтек вызовов: ");
         exc.printStackTrace();
      }
      System.out.println("После инструкции catch");
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
До генерации исключения
Стандартное сообщение:
java.lang.ArrayIndexOutOfBoundsException: 7

Стек вызовов:
java.lang.ArrayIndexOutOfBoundsException: 7
      at ExcTest.genException(UseThrowableMethods.java:10)
      at UseThrowableMethods.main(UseThrowableMethods.java:19)
После инструкции catch
```

Использование ключевого слова finally

Иногда требуется определить блок кода, который должен выполняться по завершении блока try/catch. Допустим, в процессе работы программы возникло исключение, требующее ее преждевременного завершения. Но в программе открыт файл или установлено сетевое соединение, а следовательно, файл нужно закрыть, а соединение — разорвать. Для выполнения подобных операций, связанных с нормальным завершением программы, удобно воспользоваться ключевым словом finally.

Для того чтобы определить код, который должен выполняться по завершении блока try/catch, нужно указать блок finally в конце последовательности инструкций try/catch. Ниже приведена общая форма записи блока try/catch с блоком finally.

```
try {
   // Блок кода, в котором отслеживаются ошибки
}
catch (тип_исключения_1 объект_исключения) {
   // Обработчик исключения тип_исключения_1
}
catch (тип_исключения_2 объект_исключения) {
   // Обработчик исключения тип_исключения_2
}

//...
finally {
   // Код блока finally
}
```

Блок finally выполняется всегда по завершении блока try/catch, независимо от того, какое именно условие к этому привело. Следовательно, блок finally получит управление как при нормальной работе программы, так и при возникновении ошибки. Более того, он будет вызван даже в том случае, если в блоке try или в одном из блоков catch будет стоять инструкция return, используемая для немедленного возврата из метода.

Ниже приведен краткий пример программы, демонстрирующий использование блока finally.

```
// Использование блока finally
class UseFinally {
   public static void genException(int what) {
      int t;
      int nums[] = new int[2];

      System.out.println("Получено: " + what);
      try {
         switch(what) {
            case 0:
               t = 10 / what; // сгенерировать ошибку деления
                              // на нуль
               break;
            case 1:
               nums[4] = 4; // сгенерировать ошибку обращения
                            // к массиву
               break;
            case 2:
               return; // возврат из блока try
         }
      }
```

```
        catch (ArithmeticException exc) {
            // Перехват исключения
            System.out.println("Попытка деления на нуль");
            return; // возврат из блока catch
        }
        catch (ArrayIndexOutOfBoundsException exc) {
            // перехват исключения
            System.out.println("Соответствующий элемент не найден");
        }
        finally { ◄────── Этот блок выполняется независимо от того, каким образом завершается блок try/catch
            System.out.println("Выход из блока try");
        }
    }
}

class FinallyDemo {
    public static void main(String args[]) {
        for(int i=0; i < 3; i++) {
            UseFinally.genException(i);
            System.out.println();
        }
    }
}
```

В результате выполнения данной программы будет получен следующий результат.

```
Получено: 0
Попытка деления на нуль
Выход из блока try

Получено: 1
Соответствующий элемент не найден
Выход из блока try

Получено: 2
Выход из блока try
```

Как видите, блок `finally` выполняется независимо от того, каким образом завершается блок `try/catch`.

Использование ключевого слова throws

Иногда исключения нецелесообразно обрабатывать в том методе, в котором они возникают. В таком случае их следует указывать с помощью ключевого слова `throws`. Ниже приведена общая форма объявления метода, в котором применяется ключевое слово `throws`.

```
возвращаемый_тип имя_метода(список_параметров)
    throws список_исключений {
    // Тело метода
}
```

Здесь *список_исключений* представляет собой разделенный запятыми список, в котором указываются исключения, генерируемые методом.

Возможно, вам покажется странным, что в ряде предыдущих примеров ключевое слово throws не указывалось при генерировании исключений за пределами методов. Дело в том, что исключения, генерируемые подклассом Error или RuntimeException, можно не указывать в списке throws. Исполняющая среда Java по умолчанию предполагает, что метод может их генерировать. Возможные исключения всех остальных типов ***нужно*** объявить с помощью ключевого слова throws. Если этого не сделать, возникнет ошибка во время компиляции программы.

Пример использования спецификации throws уже был представлен ранее. Напомним, что при организации ввода с клавиатуры с помощью метода main() потребовалось включить следующее выражение:

```
throws java.io.IOException
```

Теперь вы знаете, зачем это было нужно. При вводе данных может возникнуть исключение IOException, а на тот момент вы еще не знали, как оно обрабатывается. Поэтому мы и указали, что исключение должно обрабатываться за пределами метода main(). Теперь, ознакомившись с исключениями, вы сможете без труда обработать исключение IOException самостоятельно.

Рассмотрим пример, в котором обрабатывается исключение IOException. В методе prompt() отображается сообщение, а затем выполняется ввод символов с клавиатуры. Подобный ввод данных может привести к возникновению исключения IOException. Но это исключение не обрабатывается в методе prompt(). Вместо этого в объявлении метода указана инструкция throws, т.е. обязанности по обработке данного исключения поручаются вызывающему методу. В данном случае вызывающим является метод main(), в котором и перехватывается исключение.

```
// Использование ключевого слова throws
class ThrowsDemo {
  public static char prompt(String str)
    throws java.io.IOException { ◄—— Обратите внимание на спецификацию throws в объявлении метода

    System.out.print(str + ": ");
    return (char) System.in.read();
  }

  public static void main(String args[]) {
    char ch;

    try {
      ch = prompt("Введите букву"); ◄—— В методе prompt() может быть сгенерировано исключение, поэтому его вызов следует заключить в блок try
    }
    catch(java.io.IOException exc) {
      System.out.println("Произошло исключение ввода-вывода");
```

```
            ch = 'X';
        }

        System.out.println("Вы нажали клавишу " + ch);
    }
}
```

Обратите внимание на одну особенность приведенного выше примера. Класс IOException относится к пакету java.io. Как будет показано в главе 10, в этом пакете содержатся многие языковые средства Java для организации ввода-вывода. Следовательно, можно импортировать пакет java.io, а в программе указать только имя класса IOException.

Три дополнительных средства обработки исключений

С появлением версии JDK 7 механизм обработки исключений в Java был значительно усовершенствован благодаря включению в него трех новых средств. Первое из них поддерживает *автоматическое управление ресурсами*, позволяющее автоматизировать процесс освобождения таких ресурсов, как файлы, когда они больше не нужны. В основу этого средства положена расширенная форма инструкции try, называемая *инструкцией try с ресурсами* и описываемая в главе 10 при рассмотрении файлов. Второе новое средство называется *групповым перехватом*, а третье — *окончательным*, или *уточненным, повторным генерированием исключений*. Два последних средства рассматриваются ниже.

Групповой перехват позволяет перехватывать два или более исключения одной инструкцией catch. Как уже рассматривалось, после инструкции try можно (и даже принято) указывать две или более инструкции catch. И хотя каждый блок catch, как правило, содержит свой собственный код, нередко в двух или более блоках catch выполняется один и тот же код, несмотря на то что в них перехватываются разные исключения. Вместо того чтобы перехватывать каждый тип исключения в отдельности, теперь можно воспользоваться единым блоком catch для обработки исключений, тем самым избегая дублирования кода.

Для организации группового перехвата следует указать список исключений в одной инструкции catch, разделив их названия побитовым оператором ИЛИ. Каждый параметр группового перехвата неявно указывается как final. (По желанию модификатор доступа final можно указать и явным образом, но это совсем не обязательно.) А поскольку каждый параметр группового перехвата неявно указывается как final, ему нельзя присвоить новое значение.

В приведенной ниже строке кода показано, каким образом групповой перехват исключений ArithmeticException и ArrayIndexOutOfBoundsException указывается в одной инструкции catch.

```
catch(final ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

Ниже приведен пример программы, демонстрирующий применение группового перехвата исключений.

```
// Использование средства группового перехвата исключений.
// Примечание: для компиляции этого кода требуется JDK 7 или выше.
class MultiCatch {
   public static void main(String args[]) {
      int a=88, b=0;
      int result;
      char chrs[] = { 'A', 'B', 'C' };

      for(int i=0; i < 2; i++) {
         try {
            if(i == 0)
               result = a / b; // Генерирование исключения
                               // ArithmeticException
            else
               chrs[5] = 'X'; // Генерирование исключения
                              // ArrayIndexOutOfBoundsException
         }
         // В этой инструкции catch организуется перехват
         // обоих исключений
         catch(ArithmeticException |
               ArrayIndexOutOfBoundsException e) {
            System.out.println("Перехваченное исключение: " + e);
         }
      }

      System.out.println("После группового перехватчика исключений");
   }
}
```

В данном примере программы исключение `ArithmeticException` генерируется при попытке деления на нуль, а исключение `ArrayIndexOutOfBoundsException` — при попытке обращения за пределы массива `chrs`. Оба исключения перехватываются одной инструкцией `catch`.

Средство уточненного повторного генерирования исключений ограничивает этот процесс лишь теми проверяемыми типами исключений, которые генерируются в соответствующем блоке `try` и не обрабатываются в предыдущем блоке `catch`, а также относятся к подтипу или супертипу указываемого параметра. И хотя такая возможность требуется нечасто, ничто не мешает теперь воспользоваться ею в полной мере. А для организации окончательного повторного генерирования исключений параметр инструкции `catch` должен иметь модификатор доступа `final`. Это означает, что ему нельзя присвоить новое значение в блоке `catch`. Он может быть указан как `final` явным образом, хотя это и не обязательно.

Встроенные классы исключений Java

В стандартном пакете java.lang определены некоторые классы, представляющие стандартные исключения Java. Часть из них использовалась в предыдущих примерах программ. Наиболее часто встречаются исключения из подклассов стандартного класса RuntimeException. А поскольку пакет java.lang импортируется по умолчанию во все программы на Java, то исключения, производные от класса RuntimeException, становятся доступными автоматически. Их даже не обязательно включать в список throws. В терминологии Java такие исключения называются ***непроверяемыми***, поскольку компилятор не проверяет, обрабатываются или генерируются подобные исключения в методе. Непроверяемые исключения, определенные в пакете java.lang, приведены в табл. 9.2, тогда как в табл. 9.3 указаны те исключения из пакета java.lang, которые следует обязательно включать в список throws при объявлении метода, если, конечно, в методе содержатся инструкции, способные генерировать эти исключения, а их обработка не предусмотрена в теле метода. Такие исключения принято называть ***проверяемыми***. В Java предусмотрен также ряд других исключений, определения которых содержатся в различных библиотеках классов. К их числу можно отнести упомянутое ранее исключение IOException.

Таблица 9.2. Непроверяемые исключения, определенные в пакете java.lang

Исключение	Описание
ArithmeticException	Арифметическая ошибка, например попытка деления на нуль
ArrayIndexOutOfBoundsException	Попытка обращения за пределы массива
ArrayStoreException	Попытка ввести в массив элемент, несовместимый с ним по типу
ClassCastException	Недопустимое приведение типов
EnumConstNotPresentException	Попытка использования нумерованного значения, которое не было определено ранее
IllegalArgumentException	Недопустимый параметр при вызове метода
IllegalCallerException	Метод невозможно вызвать с помощью вызывающего кода корректным образом (появилось в JDK 9)
IllegalMonitorStateException	Недопустимая операция контроля, например ожидание разблокировки потока
IllegalStateException	Недопустимое состояние среды выполнения или приложения
IllegalThreadStateException	Запрашиваемая операция несовместима с текущим состоянием потока

Окончание табл. 9.2

Исключение	Описание
`IndexOutOfBoundsException`	Недопустимое значение индекса
`LayerInstantiationException`	Невозможно создать уровень модуля (появилось в JDK 9)
`NegativeArraySizeException`	Создание массива отрицательного размера
`NullPointerException`	Недопустимое использование пустой ссылки
`NumberFormatException`	Неверное преобразование символьной строки в число
`SecurityException`	Попытка нарушить систему защиты
`StringIndexOutOfBounds`	Попытка обращения к символьной строке за ее пределами
`TypeNotPresentException`	Неизвестный тип
`UnsupportedOperationException`	Неподдерживаемая операция

Таблица 9.3. Проверяемые исключения, определенные в пакете `java.lang`

Исключение	Описание
`ClassNotFoundException`	Класс не найден
`CloneNotSupportedException`	Попытка клонирования объекта, не реализующего интерфейс `Cloneable`
`IllegalAccessException`	Доступ к классу запрещен
`InstantiationException`	Попытка создания объекта абстрактного класса или интерфейса
`InterruptedException`	Прерывание одного потока другим
`NoSuchFieldException`	Требуемое поле не существует
`NoSuchMethodException`	Требуемый метод не существует
`ReflectiveOperationException`	Суперкласс исключений, связанных с рефлексией

Создание подклассов, производных от класса `Exception`

Несмотря на то что встроенные в Java исключения позволяют обрабатывать большинство ошибок, механизм обработки исключений не ограничивается только этими ошибками. В частности, можно создавать исключения для обработки потенциальных ошибок в прикладной программе. Создать исключение несложно. Для этого достаточно определить подкласс, производный от класса `Exception`, который, в свою очередь, представляет собой подкласс,

Спросим у эксперта

ВОПРОС. Говорят, что в языке Java поддерживаются специальные исключения, называемые цепочечными. Что это такое?

ОТВЕТ. Цепочечные исключения — сравнительно недавнее дополнение Java (появилось в JDK 1.4). Это языковое средство позволяет указывать одно исключение как причину появления другого. Представьте себе ситуацию, когда метод генерирует исключение `ArithmeticException` как реакцию на попытку деления на нуль. Но на самом деле в программе возникает ошибка ввода-вывода, из-за которой делитель устанавливается неверно. Поэтому было бы желательно уведомить вызывающую часть программы, что истинной причиной служит не попытка деления на нуль, а ошибка ввода-вывода, хотя исключение `ArithmeticException` безусловно должно быть сгенерировано. И это позволяют сделать цепочечные исключения. Они применимы и в других ситуациях, когда имеют место многоуровневые исключения.

Для поддержки цепочечных исключений в класс `Throwable` введены два конструктора и два метода. Ниже приведены общие формы объявления обоих конструкторов.

```
Throwable(Throwable причинное_исключение)
Throwable(String сообщение, Throwable причинное_исключение)
```

В первой форме конструктора *причинное_исключение* обозначает предыдущее исключение, послужившее причиной текущего исключения. Вторая форма конструктора позволяет указывать не только *причинное_исключение*, но и сообщение. Эти же конструкторы были введены в классы `Error`, `Exception` и `RuntimeException`.

Помимо конструкторов, в классе `Throwable` были также определены методы `getCause()` и `initCause()`. Ниже приведены общие формы объявления этих методов.

```
Throwable getCause()
Throwable initCause(Throwable причинное_исключение)
```

Метод `getCause()` возвращает исключение, которое стало причиной текущего исключения. Если такого исключения не было, возвращается пустое значение `null`. А метод `initCause()` связывает *причинное_исключение* с тем исключением, которое должно быть сгенерировано, возвращая ссылку на него. Подобным способом можно связать причину с исключением уже после того, как исключение было сгенерировано. Как правило, метод `initCause()` применяется для установления истинной причины исключения, которое сгенерировано устаревшими классами, не поддерживающими описанные выше конструкторы.

Цепочечные исключения требуются далеко не в каждой программе. Но в тех случаях, когда необходимо выяснить истинную причину исключения, это языковое средство позволяет легко решить подобную задачу.

порожденный классом Throwable. В создаваемый подкласс не обязательно включать реализацию каких-то методов. Сам факт существования такого подкласса позволяет использовать его в качестве исключения.

В классе Exception не определены новые методы. Он лишь наследует методы, предоставляемые классом Throwable. Таким образом, все исключения, включая и создаваемые вами, содержат методы класса Throwable. Конечно же, вы вольны переопределить в создаваемом вами классе один или несколько методов.

Ниже приведен пример, в котором создается исключение NonIntResult Exception. Оно генерируется в том случае, если результатом деления двух целых чисел является дробное число. В классе NonIntResultException содержатся два поля, предназначенные для хранения целых чисел, а также конструктор. В нем также переопределен метод toString(), что дает возможность выводить описание исключения с помощью метода println().

```
// Использование специально создаваемого исключения

// Создание исключения
class NonIntResultException extends Exception {
   int n;
   int d;

   NonIntResultException(int i, int j) {
      n = i;
      d = j;
   }

   public String toString() {
      return "Результат операции " + n + " / " + d +
             " не является целым числом";
   }
}

class CustomExceptDemo {
   public static void main(String args[]) {

      // В массиве numer содержатся нечетные числа
      int numer[] = { 4, 8, 15, 32, 64, 127, 256, 512 };
      int denom[] = { 2, 0, 4, 4, 0, 8 };

      for(int i=0; i<numer.length; i++) {
         try {
            if((numer[i]%2) != 0)
               throw new NonIntResultException(numer[i], denom[i]);

            System.out.println(numer[i] + " / " +
                               denom[i] + " равно " +
                               numer[i]/denom[i]);
         }
```

```
        catch (ArithmeticException exc) {
           // Перехват исключения
           System.out.println("Попытка деления на нуль");
        }
        catch (ArrayIndexOutOfBoundsException exc) {
        // Перехват исключения
        System.out.println("Соответствующий элемент не найден");
     }
     catch (NonIntResultException exc) {
        System.out.println(exc);
        }
     }
  }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
4 / 2 равно 2
Попытка деления на нуль
Результат операции 15 / 4 не является целым числом
32 / 4 равно 8
Попытка деления на нуль
Результат операции 127 / 8 не является целым числом
Соответствующий элемент не найден
Соответствующий элемент не найден
```

Спросим у эксперта

ВОПРОС. Когда следует предусматривать обработку исключений в программе? И в каких случаях имеет смысл самостоятельно определять классы исключений?

ОТВЕТ. Поскольку в Java API для формирования сообщений об ошибках широко применяются исключения, обработчики исключений должны быть практически во всех прикладных программах. Ответ на вопрос, следует ли в конкретном случае обрабатывать исключения, практически очевиден. Сложнее принять решение о том, следует ли определить собственное исключение. Обычно сообщения об ошибках формируются двумя способами: путем возврата специальных кодов и с помощью исключений. Какому из этих способов отдать предпочтение? Традиционный для программирования на Java подход состоит в использовании исключений. Разумеется, полностью отказываться от обработки возвращаемых кодов ошибок не стоит. Ведь в некоторых случаях такой способ оказывается очень удобным. Но исключения предоставляют более эффективный и структурированный механизм обработки ошибок. Именно так профессионалы организуют в своих программах реакцию на потенциальные ошибки.

Упражнение 9.1

Добавление исключений в класс очереди

```
QueueFullException.java
QueueEmptyException.java
FixedQueue.java
QExcDemo.java
```

В этом упражнении нам предстоит создать два класса исключений, которые будут использоваться классом очереди, разработанным в упражнении 8.1. Эти исключения должны указывать на переполнение и опустошение очереди, а генерировать их будут методы `put()` и `get()` соответственно. Ради простоты эти исключения добавляются в класс `FixedQueue`, но вы сможете без труда внедрить их в любые другие классы очереди, разработанные в упражнении 8.1. Поэтапное описание процесса создания программы приведено ниже.

1. Вам предстоит создать два файла, которые будут включать классы исключений очереди. Присвойте первому файлу имя `QueueFullException.java` и введите в него следующий код.

```
// Исключение для ошибок, связанных с заполненной очередью
public class QueueFullException extends Exception {
   int size;

   QueueFullException(int s) { size = s; }

   public String toString() {
      return "\nОчередь заполнена. Максимальный размер " + size;
   }
}
```

   В случае попытки сохранить элемент в уже заполненной очереди генерируется исключение `QueueFullException`.

2. Создайте второй файл `QueueEmptyException.java` и введите в него следующий код.

```
// Исключение для ошибок, связанных с пустой очередью
public class QueueEmptyException extends Exception {

   public String toString() {
      return "\nОчередь пуста.";
   }
}
```

   Исключение `QueueEmptyException` генерируется в случае попытки удалить элемент из пустой очереди.

3. Измените класс `FixedQueue` таким образом, чтобы генерировались исключения в случае ошибок, как показано здесь. Введите этот код в файл `FixedQueue.java`.

```
// Класс очереди фиксированного размера для символов,
// использующий исключения
class FixedQueue implements ICharQ {
   private char q[];// массив для хранения элементов очереди
```

```
    private int putloc, getloc; // индексы вставляемых и
                                // извлекаемых элементов

    // Создание пустой очереди заданного размера
    public FixedQueue(int size) {
        q = new char[size]; // выделение памяти для очереди
        putloc = getloc = 0;
    }

    // Помещение символа в очередь
    public void put(char ch) throws QueueFullException {

        if(putloc==q.length)
            throw new QueueFullException(q.length);

        q[putloc++] = ch;
    }

    // Извлечение символа из очереди
    public char get() throws QueueEmptyException {

        if(getloc == putloc)
            throw new QueueEmptyException();

        return q[getloc++];
    }
}
```

Добавление исключений в класс `FixedQueue` осуществляется в два этапа. Сначала в определение методов `get()` и `put()` добавляется ключевое слово `throws` с названием генерируемого исключения. Затем в этих методах организуется генерирование исключений при возникновении ошибок. Используя исключения, можно организовать обработку ошибок в вызывающей части программы наиболее рациональным способом. Как вы помните, в предыдущих версиях рассматриваемой здесь программы выводились только сообщения об ошибках. Однако генерирование исключений является более профессиональным подходом к разработке данной программы.

4. В качестве самостоятельного эксперимента с усовершенствованным классом `FixedQueue` введите в файл `QExcDemo.java` приведенный ниже исходный код класса `QExcDemo`.

```
// Демонстрация исключений при работе с очередью
class QExcDemo {
    public static void main(String args[]) {
        FixedQueue q = new FixedQueue(10);
        char ch;
        int i;

        try {
            // Переполнение очереди
```

```
            for(i=0; i < 11; i++) {
                System.out.print("Попытка сохранения: " +
                                (char) ('A' + i));
                q.put((char) ('A' + i));
                System.out.println(" - OK");
            }
            System.out.println();
        }
        catch (QueueFullException exc) {
            System.out.println(exc);
        }
        System.out.println();

        try {
            // Попытка извлечения символа из пустой очереди
            for(i=0; i < 11; i++) {
                System.out.print("Получение очередного символа: ");
                ch = q.get();
                System.out.println(ch);
            }
        }
        catch (QueueEmptyException exc) {
            System.out.println(exc);
        }
    }
}
```

5. Класс `FixedQueue` реализует интерфейс `ICharQ`, в котором определены методы `get()` и `put()`, и поэтому интерфейс `ICharQ` необходимо изменить таким образом, чтобы в нем отражалось наличие спецификаций `throws`. Ниже приведен видоизмененный код интерфейса `ICharQ`. Не забывайте о том, что он должен храниться в файле `ICharQ.java`.

```
// Интерфейс очереди для хранения символов с генерированием исключений
public interface ICharQ {
    // Помещение символа в очередь
    void put(char ch) throws QueueFullException;

    // Извлечение символа из очереди
    char get() throws QueueEmptyException;
}
```

6. Скомпилируйте сначала новую версию исходного файла `IQChar.java`, а затем исходный файл `QExcDemo.java` и запустите программу `QExcDemo` на выполнение. В итоге будет получен следующий результат.

```
Попытка сохранения: A - OK
Попытка сохранения: B - OK
Попытка сохранения: C - OK
Попытка сохранения: D - OK
Попытка сохранения: E - OK
Попытка сохранения: F - OK
```

```
Попытка сохранения: G - OK
Попытка сохранения: H - OK
Попытка сохранения: I - OK
Попытка сохранения: J - OK
Попытка сохранения: K
Очередь заполнена. Максимальный размер очереди: 10

Получение очередного символа: A
Получение очередного символа: B
Получение очередного символа: C
Получение очередного символа: D
Получение очередного символа: E
Получение очередного символа: F
Получение очередного символа: G
Получение очередного символа: H
Получение очередного символа: I
Получение очередного символа: J
Получение очередного символа:
Очередь пуста.
```

Вопросы и упражнения для самопроверки

1. Какой класс находится на вершине иерархии исключений?
2. Объясните вкратце, как используются ключевые слова `try` и `catch`?
3. Какая ошибка допущена в приведенном ниже фрагменте кода?

```
// ...
vals[18] = 10;
catch (ArrayIndexOutOfBoundsException exc) {
   // обработка ошибки
}
```

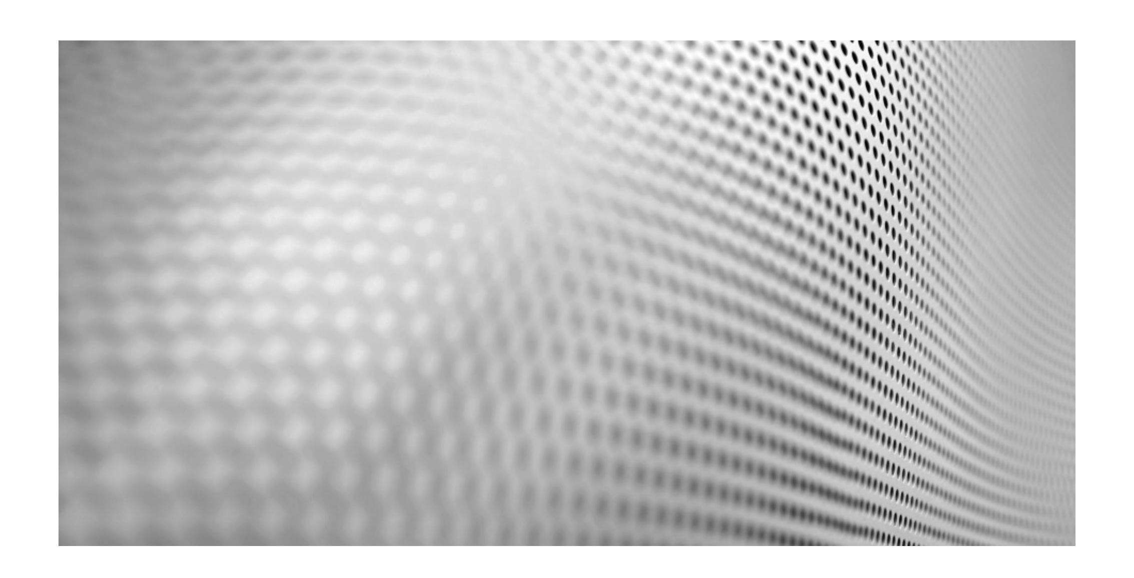
4. Что произойдет, если исключение не будет перехвачено?
5. Какая ошибка допущена в приведенном ниже фрагменте кода?

```
class A extends Exception { ...

class B extends A { ...

// ...

try {
   // ...
}
catch (A exc) { ... }
catch (B exc) { ... }
```

6. Может ли внутренний блок `catch` повторно сгенерировать исключение, которое будет обработано во внешнем блоке `catch`?
7. Блок `finally` — последний фрагмент кода, выполняемый перед завершением программы. Верно или неверно? Обоснуйте свой ответ.
8. Исключения какого типа необходимо явно объявлять с помощью инструкции `throws`, включаемой в объявление метода?
9. Какая ошибка допущена в приведенном ниже фрагменте кода?

```
class MyClass { // ... }
// ...
throw new MyClass();
```

10. Отвечая на вопрос 3 в конце главы 6, вы создали класс `Stack`. Добавьте в него пользовательские исключения, чтобы программа нужным образом реагировала на попытку поместить элемент в переполненный стек и извлечь элемент из пустого стека.
11. Назовите три причины, по которым могут генерироваться исключения.
12. Назовите два подкласса, производных непосредственно от класса `Throwable`.
13. Что такое групповой перехват исключений?
14. Следует ли перехватывать в программе исключения типа `Error`?



Глава 10

Ввод-вывод данных

В этой главе...

- Потоки ввода-вывода
- Отличия байтовых и символьных потоков
- Классы для поддержки байтовых потоков
- Классы для поддержки символьных потоков
- Знакомство со встроенными потоками
- Использование байтовых потоков
- Использование байтовых потоков для файлового ввода-вывода
- Автоматическое закрытие файлов с помощью инструкции `try` с ресурсами
- Чтение и запись двоичных данных
- Манипулирование файлами с произвольным доступом
- Использование символьных потоков
- Использование символьных потоков для файлового ввода-вывода
- Применение оболочек типов Java для преобразования числовых строк

В предыдущих главах уже рассматривались примеры программ, в которых использовались отдельные элементы подсистемы ввода-вывода Java, в частности, метод `println()`, но все это делалось без каких-либо формальных пояснений. В Java подсистема ввода-вывода основана на иерархии классов, и поэтому ее невозможно было рассматривать, не узнав предварительно, что такое классы, наследование и исключения. Теперь, когда вы уже в достаточной степени к этому подготовлены, мы можем приступить к обсуждению средств ввода-вывода.

Следует отметить, что подсистема ввода-вывода Java очень обширна и включает множество классов, интерфейсов и методов. Отчасти это объясняется тем, что в Java определены фактически две полноценные подсистемы ввода-вывода: одна — для обмена байтами, другая — для обмена символами. Здесь нет возможности рассмотреть все аспекты ввода-вывода в Java, ведь для этого потребовалась бы отдельная книга. Поэтому в данной главе будут рассмотрены лишь наиболее важные и часто используемые языковые средства ввода-вывода. Правда, элементы подсистемы ввода-вывода в Java тесно взаимосвязаны, и поэтому, уяснив основы, вы легко освоите все остальные нюансы этой подсистемы.

Прежде чем приступать к рассмотрению подсистемы ввода-вывода, необходимо сделать следующее замечание. Классы, описанные в этой главе, предназначены для консольного и файлового ввода-вывода. Они не используются для

создания графических пользовательских интерфейсов. Поэтому при создании оконных приложений они вам не пригодятся. Для создания графических интерфейсов в Java предусмотрены другие средства. Они будут представлены в главах 16 и 17, где вы познакомитесь с библиотеками Swing и JavaFX соответственно.

Потоковая организация ввода-вывода в Java

В Java операции ввода-вывода реализованы на основе потоков. Поток — это абстрактная сущность, представляющая устройства ввода-вывода, которая выдает и получает информацию. За связь потоков с физическими устройствами отвечает подсистема ввода-вывода, что позволяет работать с разными устройствами, используя одни и те же классы и методы. Например, методы вывода на консоль в равной степени могут быть использованы для записи данных в дисковый файл. Для реализации потоков используется иерархия классов, содержащихся в пакете `java.io`.

Байтовые и символьные потоки

В современных версиях Java определены два типа потоков: байтовые и символьные. (Первоначально в Java были доступны только байтовые потоки, но вскоре были реализованы и символьные.) Байтовые потоки предоставляют удобные средства для управления вводом и выводом байтов. Например, их можно использовать для чтения и записи двоичных данных. Потоки этого типа особенно удобны при работе с файлами. С другой стороны, символьные потоки ориентированы на обмен символьными данными. В них применяется кодировка Unicode, и поэтому их легко интернационализировать. Кроме того, в некоторых случаях символьные потоки более эффективны по сравнению с байтовыми потоками.

Необходимость поддерживать два разных типа потоков ввода-вывода привела к созданию двух иерархий классов: одна — для байтовых, другая — для символьных данных. Из-за того что число классов достаточно велико, на первый взгляд подсистема ввода-вывода кажется сложнее, чем она есть на самом деле. Просто знайте, что в большинстве случаев функциональные возможности символьных потоков идентичны возможностям байтовых.

Вместе с тем на самом нижнем уровне все средства ввода-вывода имеют байтовую организацию. Символьные потоки лишь предоставляют удобные и эффективные средства, адаптированные к специфике обработки символов.

Классы байтовых потоков

Байтовые потоки определены с использованием двух иерархий классов, на вершинах которых находятся абстрактные классы `InputStream` и `OutputStream` соответственно. В классе `InputStream` определены свойства, общие для байтовых потоков ввода, а в классе `OutputStream` — свойства, общие для байтовых потоков вывода.

Производными от классов InputStream и OutputStream являются конкретные подклассы, реализующие различные функциональные возможности и учитывающие особенности обмена данными с разными устройствами, например ввода-вывода в файлы на диске. Классы байтовых потоков перечислены в табл. 10.1. Пусть вас не пугает большое количество этих классов: изучив один из них, вы легко освоите остальные.

Таблица 10.1. Классы байтовых потоков

Класс байтового потока	Описание
BufferedInputStream	Буферизованный входной поток
BufferedOutputStream	Буферизованный выходной поток
ByteArrayInputStream	Входной поток для чтения из байтового массива
ByteArrayOutputStream	Выходной поток для записи в байтовый массив
DataInputStream	Входной поток, включающий методы для чтения стандартных типов данных Java
DataOutputStream	Выходной поток, включающий методы для записи стандартных типов данных Java
FileInputStream	Входной поток для чтения из файла
FileOutputStream	Выходной поток для записи в файл
FilterInputStream	Реализация класса InputStream
FilterOutputStream	Реализация класса OutputStream
InputStream	Абстрактный класс, описывающий потоковый ввод
ObjectInputStream	Входной поток для объектов
ObjectOutputStream	Выходной поток для объектов
OutputStream	Абстрактный класс, описывающий потоковый вывод
PipedInputStream	Входной канал
PipedOutputStream	Выходной канал
PrintStream	Выходной поток, включающий методы print() и println()
PushbackInputStream	Входной поток, позволяющий возвращать байты обратно в поток
SequenceInputStream	Входной поток, сочетающий в себе несколько потоков, которые читаются последовательно, один после другого

Классы символьных потоков

Символьные потоки также определены с использованием двух иерархий классов, вершины которых на этот раз представлены абстрактными классами Reader и Writer соответственно. Класс Reader и его подклассы используются для чтения, а класс Writer и его подклассы — для записи данных. Конкретные

классы, производные от классов Reader и Writer, оперируют символами в кодировке Unicode.

Классы, производные от классов Reader и Writer, предназначены для выполнения различных операций ввода-вывода символов. В целом символьные классы представляют собой аналоги соответствующих классов, предназначенных для работы с байтовыми потоками. Классы символьных потоков перечислены в табл. 10.2.

Таблица 10.2. Классы символьных потоков

Класс символьного потока	Описание
BufferedReader	Буферизованный входной символьный поток
BufferedWriter	Буферизованный выходной символьный поток
CharArrayReader	Входной поток для чтения из символьного массива
CharArrayWriter	Выходной поток для записи в символьный массив
FileReader	Входной поток для чтения из файла
FileWriter	Выходной поток для записи в файл
FilterReader	Фильтрующий входной поток
FilterWriter	Фильтрующий выходной поток
InputStreamReader	Входной поток, транслирующий байты в символы
LineNumberReader	Входной поток, подсчитывающий строки
OutputStreamWriter	Выходной поток, транслирующий символы в байты
PipedReader	Входной канал
PipedWriter	Выходной канал
PrintWriter	Выходной поток, включающий методы print() и println()
PushbackReader	Входной поток, позволяющий возвращать символы обратно в поток
Reader	Абстрактный класс, описывающий символьный ввод
StringReader	Входной поток для чтения из строки
StringWriter	Выходной поток для записи в строку
Writer	Абстрактный класс, описывающий символьный вывод

Встроенные потоки

Как вы уже знаете, во все программы на Java автоматически импортируется пакет java.lang, в котором определен класс System, инкапсулирующий некоторые свойства среды выполнения. Помимо прочего, в нем содержатся предопределенные переменные in, out и err, представляющие стандартные потоки ввода-вывода. Эти поля объявлены как public, final и static, т.е. к ним можно обращаться из любой другой части программы, не ссылаясь на конкретный объект типа System.

Переменная `System.out` ссылается на стандартный выходной поток, который по умолчанию связан с консолью. Переменная `System.in` ссылается на стандартный входной поток, который по умолчанию связан с клавиатурой. И наконец, переменная `System.err` ссылается на стандартный поток ошибок, который, как и выходной поток, также связан по умолчанию с консолью. При необходимости каждый из этих потоков может быть перенаправлен на любое другое устройство.

Поток `System.in` — это объект типа `InputStream`, а потоки `System.out` и `System.err` — объекты типа `PrintStream`. Все эти потоки — байтовые, хотя обычно они используются для чтения и записи символов с консоли и на консоль. Дело в том, что в первоначальной спецификации Java, в которой символьные потоки вообще отсутствовали, все предопределенные потоки были байтовыми. Как вы далее увидите, по мере необходимости их можно поместить в классы-оболочки символьных потоков.

Использование байтовых потоков

Начнем рассмотрение подсистемы ввода-вывода в Java с байтовых потоков. Как уже отмечалось, на вершине иерархии байтовых потоков находятся классы `InputStream` и `OutputStream`. Методы класса `InputStream` перечислены в табл. 10.3, а методы класса `OutputStream` — в табл. 10.4. При возникновении ошибок во время выполнения методы классов `InputStream` и `OutputStream` могут генерировать исключения `IOException`. Определенные в этих двух абстрактных классах методы доступны во всех подклассах. Таким образом, они образуют минимальный набор функций ввода-вывода, общий для всех байтовых потоков.

Таблица 10.3. Методы, определенные в классе `InputStream`

Метод	Описание
`int available()`	Возвращает количество байтов ввода, доступных в данный момент для чтения
`void mark(int numBytes)`	Помещает в текущую позицию входного потока метку, которая будет находиться там до тех пор, пока не будет прочитано количество байтов, определяемое параметром *numBytes*
`boolean markSupported()`	Возвращает значение `true`, если методы `mark()` и `reset()` поддерживаются вызывающим потоком
`int read()`	Возвращает целочисленное представление следующего байта в потоке. По достижении конца потока возвращается значение –1

Окончание табл. 10.3

Метод	Описание
`int read(byte buffer[])`	Пытается прочитать *buffer.length* байтов в массив *buffer*, возвращая фактическое количество успешно прочитанных байтов. По достижении конца потока возвращается значение –1
`int read(byte buffer[], int offset, int numBytes)`	Пытается прочитать *buffer.length* байтов в массив *buffer*, начиная с элемента *buffer[offset]*, и возвращает фактическое количество успешно прочитанных байтов. По достижении конца потока возвращается значение –1
`byte[] readAllBytes()`	Считывает и возвращает в виде байтового массива все байты, доступные в потоке. В результате попытки считывания конца потока будет создан пустой массив (добавлено в JDK 9)
`int readNBytes(byte buffer[], int offset, int numBytes)`	Попытка считывания *numBytes* байтов в буфер *buffer*, начинающийся с *buffer[offset]*, возвращая количество успешно считанных байтов. В результате попытки считывания конца потока будет прочитано 0 байтов (добавлено в JDK 9)
`void reset()`	Сбрасывает входной указатель на ранее установленную метку
`long skip (long numBytes)`	Пропускает *numBytes* входных байтов, возвращая фактическое количество пропущенных байтов
`long transferTo(OutputStream outStrm)`	Копирует содержимое вызывающего потока в *outStrm*, возвращая количество скопированных байтов (добавлено в JDK 9)

Таблица 10.4. Методы, определенные в классе `OutputStream`

Метод	Описание
`void close()`	Закрывает выходной поток. Дальнейшие попытки записи будут генерировать исключение `IOException`
`void flush()`	Выполняет принудительную передачу содержимого выходного буфера в место назначения (тем самым очищая выходной буфер)
`void write(int b)`	Записывает один байт в выходной поток. Обратите внимание на то, что параметр имеет тип `int`, что позволяет вызывать

Окончание табл. 10.4

Метод	Описание
	метод write() с выражениями, не приводя их к типу byte
void write(byte *buffer*[])	Записывает полный массив байтов в выходной поток
void write(byte *buffer*[], int *offset*, int *numBytes*)	Записывает часть массива *buffer* в количестве *numBytes* байтов, начиная с элемента buffer[*offset*]

Консольный ввод

Первоначально байтовые потоки были единственным средством, позволяющим выполнять консольный ввод, и во многих существующих программах на Java для этой цели по-прежнему используются исключительно байтовые потоки. Сейчас имеется возможность выбора между байтовыми и символьными потоками.

В коммерческом коде для чтения консольного ввода предпочтительнее использовать символьные потоки. Такой подход упрощает интернационализацию программ и облегчает их сопровождение. Ведь намного удобнее оперировать непосредственно символами, не тратя время и усилия на преобразование символов в байты и наоборот. Однако в простых служебных и прикладных программах, где данные, введенные с клавиатуры, обрабатываются непосредственно, удобно пользоваться байтовыми потоками. Именно по этой причине они здесь и рассматриваются.

Поток System.in является экземпляром класса InputStream, и благодаря этому обеспечивается автоматический доступ к методам, определенным в классе InputStream. К сожалению, для чтения байтов в классе InputStream определен только один метод ввода: read(). Ниже приведены три возможные формы объявления этого метода.

```
int read() throws IOException
int read(byte data[]) throws IOException
int read(byte data[], int start, int max) throws IOException
```

В главе 3 было продемонстрировано, как пользоваться первой формой метода read() для чтения отдельных символов с клавиатуры (а по сути, из потока стандартного ввода System.in). Достигнув конца потока, этот метод возвращает значение –1. Вторая форма метода read() предназначена для чтения данных из входного потока в массив data. Чтение завершается по достижении конца потока, при заполнении массива или возникновении ошибки. Метод возвращает количество прочитанных байтов или –1, если достигнут конец потока. И третья форма данного метода позволяет разместить прочитанные данные в массиве *data*, начиная с элемента, заданного с помощью индекса *start*. Максимальное количество байтов, которые могут быть введены в массив,

определяется параметром *max*. Метод возвращает число прочитанных байтов или значение –1, если достигнут конец потока. При возникновении ошибки в каждой из этих форм метода `read()` генерируется исключение `IOException`. Признак конца потока ввода при чтении из `System.in` устанавливается после нажатия клавиши <Enter>.

Ниже приведен пример короткой программы, демонстрирующий чтение байтов из потока ввода `System.in` в массив. Следует иметь в виду, что исключения, которые могут быть сгенерированы при выполнении данной программы, обрабатываются за пределами метода `main()`. Подобный подход часто используется при чтении данных с консоли. По мере необходимости вы сможете самостоятельно организовать обработку ошибок.

```
// Чтение байтов с клавиатуры в массив
import java.io.*;

class ReadBytes {
   public static void main(String args[])
      throws IOException {
         byte data[] = new byte[10];

         System.out.println("Введите символы.");
         System.in.read(data);  ◄——— Чтение байтового массива с клавиатуры
         System.out.print("Вы ввели: ");
         for(int i=0; i < data.length; i++)
            System.out.print((char) data[i]);
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Введите символы.
Read Bytes
Вы ввели: Read Bytes
```

Вывод на консоль

Как и в случае консольного ввода, в Java для консольного вывода первоначально были предусмотрены только байтовые потоки. Символьные потоки были добавлены в версии Java 1.1. Для переносимого кода в большинстве случаев рекомендуется использовать символьные потоки. Однако, поскольку поток `System.out` — байтовый, он по-прежнему широко используется для побайтового вывода данных на консоль. Именно такой подход до сих пор применялся в примерах, представленных в книге. Поэтому существует необходимость рассмотреть его более подробно.

Вывод данных на консоль проще всего осуществлять с помощью уже знакомых вам методов `print()` и `println()`. Эти методы определены в классе `PrintStream` (на объект данного типа ссылается переменная потока

стандартного вывода System.out). Несмотря на то что System.out является байтовым потоком, его вполне можно использовать для простого консольного вывода.

Поскольку класс PrintStream является выходным потоком, производным от класса OutputStream, он также реализует низкоуровневый метод write(), который может быть использован для записи на консоль. Ниже приведена простейшая форма метода write(), определенного в классе PrintStream:

```
void write(int byteval)
```

Данный метод записывает в файл значение байта, переданное с помощью параметра *byteval*. Несмотря на то что этот параметр объявлен как int, в нем учитываются только младшие 8 бит. Ниже приведен простой пример программы, в которой метод write() используется для вывода символа X и символа перевода строки на консоль.

```
// Демонстрация метода System.out.write()
class WriteDemo {
   public static void main(String args[]) {
      int b;

      b = 'X';
      System.out.write(b);  ◄——— Запись байта в поток
      System.out.write('\n');
   }
}
```

Вам не часто придется использовать метод write() для вывода на консоль, хотя в некоторых ситуациях он оказывается весьма кстати. Для этой цели намного удобнее пользоваться методами print() и println().

В классе PrintStream реализованы два дополнительных метода, printf() и format(), которые позволяют управлять форматированием выводимых данных. Например, они позволяют указать для выводимых данных количество десятичных цифр, минимальную ширину поля или способ представления отрицательных числовых значений. И хотя эти методы не используются в примерах, представленных в данной книге, вам стоит обратить на них пристальное внимание, поскольку они могут пригодиться при написании прикладных программ.

Чтение и запись файлов с использованием байтовых потоков

Язык Java предоставляет множество классов и методов, позволяющих читать и записывать данные из файлов и в файлы. Разумеется, чаще всего приходится обращаться к файлам, хранящимся на дисках. В Java все файлы имеют байтовую организацию, и поэтому для побайтового чтения и записи данных из файла и в файл предусмотрены соответствующие методы. Таким образом, файловые операции с использованием байтовых потоков довольно распространены.

Кроме того, для байтовых потоков ввода-вывода в файлы в Java допускается создавать оболочки в виде символьных объектов. Классы-оболочки будут рассмотрены далее.

Байтовые потоки, связанные с файлами, создаются с помощью классов `FileInputStream` или `FileOutputStream`. Чтобы открыть файл, достаточно создать объект одного из этих классов, передав конструктору имя файла в качестве параметра. Открытие файла необходимо для того, чтобы с ним можно было выполнять файловые операции чтения и записи.

Чтение данных из файла

Файл открывается для чтения путем создания объекта типа `FileInputStream`. Для этой цели чаще всего используется следующая форма конструктора данного класса:

```
FileInputStream(String имя_файла) throws FileNotFoundException
```

Имя файла, который требуется открыть, передается конструктору в параметре *имя_файла*. Если указанного файла не существует, генерируется исключение `FileNotFoundException`.

Для чтения данных из файла используется метод `read()`. Ниже приведена форма объявления этого метода, которой мы будем пользоваться в дальнейшем.

```
int read() throws IOException
```

При каждом вызове метод `read()` читает байт из файла и возвращает его в виде целочисленного значения. По достижении конца файла этот метод возвращает значение –1. При возникновении ошибки метод генерирует исключение `IOException`. Как видите, в этой форме метод `read()` выполняет те же самые действия, что и одноименный метод, предназначенный для ввода данных с консоли.

После завершения операций с файлом следует закрыть его с помощью метода `close()`, имеющего следующую общую форму объявления:

```
void close() throws IOException
```

При закрытии файла освобождаются связанные с ним системные ресурсы, которые вновь можно будет использовать для работы с другими файлами. Если этого не сделать, возможна *утечка памяти* из-за того, что часть памяти остается выделенной для ресурсов, которые больше не используются.

Ниже приведен пример программы, в которой метод `read()` используется для получения и отображения содержимого текстового файла. Имя файла указывается в командной строке при запуске программы. Обратите внимание на то, что ошибки ввода-вывода обрабатываются с помощью блока `try/catch`.

```
/* Отображение текстового файла.

   При вызове этой программы следует указать имя файла,
   содержимое которого требуется просмотреть.
   Например, для вывода на экран содержимого файла TEST.TXT
   необходимо ввести в командной строке следующую команду:
```

```
   java ShowFile TEST.TXT
*/
import java.io.*;

class ShowFile {
   public static void main(String args[])
   {
      int i;
      FileInputStream fin;

      // Сначала нужно убедиться в том, что программе
      // передается имя файла
      if(args.length != 1) {
         System.out.println("Использование: ShowFile имя_файла");
         return;
      }

      try {
         fin = new FileInputStream(args[0]);  ← Открытие файла
      } catch(FileNotFoundException exc) {
         System.out.println("Файл не найден");
         return;
      }

      try {
         // Чтение байтов, пока не встретится символ EOF
         do {
            i = fin.read();  ← Считывание данных из файла
            if(i != -1) System.out.print((char) i);
         } while(i != -1);  ← Если i =-1, значит, достигнут конец файла
      } catch(IOException exc) {
         System.out.println("Ошибка при чтении файла");
      }

      try {
         fin.close();  ← Закрытие файла
      } catch(IOException exc) {
         System.out.println("Ошибка при закрытии файла");
      }
   }
}
```

Обратите внимание на то, что в этом примере файловый поток закрывается после завершения выполнения блока `try`, в котором осуществляется чтение данных. Такой способ не всегда оказывается удобным, и поэтому в Java имеется более совершенный и чаще используемый способ, предполагающий помещение вызова метода `close()` в блок `finally`. В этом случае все методы, получающие доступ к файлу, помещаются в блок `try`, а для закрытия файла используется блок `finally`. Благодаря этому файл закрывается независимо от того, как завершится блок `try`. Учитывая это, перепишем блок `try` из предыдущего примера в таком виде.

```
try {
   do {
      i = fin.read();
      if(i != -1) System.out.print((char) i);
   } while(i != -1);
} catch(IOException exc) {
   System.out.println("Ошибка при чтении файла");
   // Для закрытия файла используется блок finally
} finally {
   // Закрыть файл при выходе из блока try
   try {
      fin.close();
   } catch(IOException exc) {
      System.out.println("Ошибка при закрытии файла");
   }
}
```

Использование блока finally для закрытия файла

Данный способ обеспечивает, в частности, то преимущество, что в случае аварийного завершения программы из-за возникновения исключения, не связанного с операциями ввода-вывода, файл все равно будет закрываться в блоке finally. И если с аварийным завершением простых программ в связи с непредвиденными исключениями, как в большинстве примеров книги, еще можно как-то мириться, то в больших программах подобная ситуация вряд ли может считаться допустимой. Использование блока finally позволяет справиться с этой проблемой.

Иногда части программы, ответственные за открытие файла и осуществление доступа к нему, удобнее поместить в один блок try (не разделяя их), а для закрытия файла использовать блок finally. В качестве примера ниже приведена измененная версия рассмотренной выше программы ShowFile.

```
/* В этой версии программы те ее части, которые отвечают
   за открытие файла и получение доступа к нему, помещены
   в один блок try. Файл закрывается в блоке finally.
*/
import java.io.*;

class ShowFile {
   public static void main(String args[])
   {
      int i;
      FileInputStream fin = null;

      // Сначала нужно убедиться в том, что программе
      // передается имя файла
      if(args.length != 1) {
         System.out.println("Использование: ShowFile имя_файла");
         return;
      }
```

Переменная fin инициализируется значением null

```
      // Открытие файла, чтение из него символов, пока
      // не встретится признак конца файла EOF, и
      // последующее закрытие файла в блоке finally
      try {
         fin = new FileInputStream(args[0]);

         do {
            i = fin.read();
            if(i != -1) System.out.print((char) i);
         } while(i != -1);

      } catch(FileNotFoundException exc) {
         System.out.println("Файл не найден.");
      } catch(IOException exc) {
         System.out.println("Ошибка ввода-вывода");
      } finally {
         // Файл закрывается в любом случае
         try {
            if(fin != null) fin.close(); ◄── Закрыть файл, если значение fin
         } catch(IOException exc) {          не равно null
            System.out.println("Ошибка при закрытии файла");
         }
      }
   }
}
```

Обратите внимание на то, что переменная `fin` инициализируется значением `null`. В блоке `finally` файл закрывается только в том случае, если значение переменной `fin` не равно `null`. Это будет работать, поскольку переменная `fin` не содержит значение `null` лишь в том случае, когда файл был успешно открыт. Следовательно, если во время открытия файла возникнет исключение, метод `close()` не будет вызываться.

В этом примере блок `try/catch` можно сделать несколько более компактным. Поскольку исключение `FileNotFoundException` является подклассом исключения `IOException`, его не нужно перехватывать отдельно. В качестве примера ниже приведен блок `catch`, которым можно воспользоваться для перехвата обоих типов исключений, избегая независимого перехвата исключения `FileNotFoundException`. В данном случае выводится стандартное сообщение о возникшем исключении с описанием ошибки.

```
...
} catch(IOException exc) {
   System.out.println("Ошибка ввода-вывода: " + exc);
} finally {
...
```

Спросим у эксперта

ВОПРОС. Метод `read()` возвращает значение –1 по достижении конца файла, но для ошибки при попытке доступа к файлу специальное возвращаемое значение не предусмотрено. Почему?

ОТВЕТ. В Java для обработки ошибок используются исключения. Поэтому если метод `read()` или любой другой возвращает конкретное значение, то это автоматически означает, что в процессе его работы ошибка не возникла. Подобный подход многие считают гораздо более удобным, чем использование специальных кодов ошибок.

При таком подходе любая ошибка, в том числе и ошибка открытия файла, будет обработана единственной инструкцией `catch`. Благодаря своей компактности в большинстве примеров ввода-вывода, представленных в этой книге, используется именно такой способ. Следует, однако, иметь в виду, что он может оказаться не вполне пригодным в тех случаях, когда требуется отдельно обрабатывать ошибку открытия файла, вызванную, например, опечаткой при вводе имени файла. В подобных случаях рекомендуется сначала предложить пользователю заново ввести имя файла, а не входить сразу же в блок `try`, в котором осуществляется доступ к файлу.

Запись в файл

Чтобы открыть файл для записи, следует создать объект типа `FileOutputStream`. Ниже приведены две наиболее часто используемые формы конструктора этого класса.

```
FileOutputStream(String имя_файла) throws FileNotFoundException
FileOutputStream(String имя_файла, boolean append)
   throws FileNotFoundException
```

В случае невозможности создания файла возникает исключение `FileNotFoundException`. Если файл с указанным именем уже существует, то в тех случаях, когда используется первая форма конструктора, этот файл удаляется. Вторая форма отличается от первой наличием параметра *`append`*. Если этот параметр имеет значение `true`, то записываемые данные добавляются в конец файла. В противном случае прежние данные в файле перезаписываются новыми.

Для записи данных в файл вызывается метод `write()`. Наиболее простая форма объявления этого метода выглядит так:

```
void write(int byteval) throws IOException
```

Данный метод записывает в поток байтовое значение, передаваемое с помощью параметра *`byteval`*. Несмотря на то что этот параметр объявлен как `int`, учитываются только младшие 8 бит его значения. Если в процессе записи возникает ошибка, генерируется исключение `IOException`.

По завершении работы с файлом его нужно закрыть с помощью метода `close()`, представленного ниже:

```
void close() throws IOException
```

При закрытии файла освобождаются связанные с ним системные ресурсы, что позволяет использовать их в дальнейшем для работы с другими файлами. Кроме того, процедура закрытия файла гарантирует, что оставшиеся в буфере данные будут записаны на диск.

В следующем примере осуществляется копирование текстового файла. Имена исходного и целевого файлов указываются в командной строке.

```
/* Копирование текстового файла.
   При вызове этой программы следует указать имена исходного
   и целевого файлов. Например, для копирования файла FIRST.TXT
   в файл SECOND.TXT в командной строке нужно ввести следующую
   команду:

   java CopyFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CopyFile {
   public static void main(String args[]) throws IOException
   {
      int i;
      FileInputStream fin = null;
      FileOutputStream fout = null;

      // Сначала нужно убедиться в том, что программе
      // передаются имена обоих файлов
      if(args.length != 2) {
         System.out.println("Использование: CopyFile -
                             источник и назначение");
         return;
      }

      // Копирование файла
      try {
         // Попытка открытия файлов
         fin = new FileInputStream(args[0]);
         fout = new FileOutputStream(args[1]);

         do {
            i = fin.read();              // Чтение байтов из одного файла
            if(i != -1) fout.write(i);   // и запись их в другой файл
         } while(i != -1);
      } catch(IOException exc) {
         System.out.println("Ошибка ввода-вывода: " + exc);
      } finally {
```

```
            try {
               if(fin != null) fin.close();
            } catch(IOException exc) {
               System.out.println("Ошибка при закрытии
                                   входного файла");
            }
            try {
               if(fout != null) fout.close();
            } catch(IOException exc) {
               System.out.println("Ошибка при закрытии выходного файла");
            }
         }
      }
   }
```

Автоматическое закрытие файлов

В примерах программ, представленных в предыдущем разделе, для закрытия файлов, которые больше не нужны, метод `close()` вызывался явным образом. Такой способ закрытия файлов используется еще с тех пор, как вышла первая версия Java. Именно поэтому он часто встречается в существующих программах. Более того, он до сих пор остается вполне оправданным и полезным. Однако в версию JDK 7 включено новое средство, предоставляющее другой, более рациональный способ управления ресурсами, в том числе и потоками файлового ввода-вывода, автоматизирующий процесс закрытия файлов. Этот способ называемый *автоматическим управлением ресурсами*, основывается на новой разновидности инструкции `try`, называемой инструкцией *`try` с ресурсами*. Главное преимущество инструкции `try` с ресурсами заключается в том, что она предотвращает ситуации, в которых файл (или другой ресурс) непреднамеренно остается неосвобожденным даже после того, как необходимость в его использовании отпала. Как пояснялось ранее, если не позаботиться о своевременном закрытии файлов, это может привести к утечке памяти и прочим осложнениям в работе программы.

Так выглядит общая форма инструкции `try` с ресурсами:

```
try (описание_ресурса) {
   // использовать ресурс
}
```

Здесь *`описание_ресурса`* включает в себя объявление и инициализацию ресурса, такого как файл. По сути, в это описание входит объявление переменной, которая инициализируется ссылкой на объект управляемого ресурса. По завершении блока `try` объявленный ресурс автоматически освобождается. Если этим ресурсом является файл, то он автоматически закрывается, что избавляет от необходимости вызывать метод `close()` явным образом. Инструкция `try` с ресурсами также может включать блоки `catch` и `finally`.

Примечание

Начиная с JDK 9 спецификация ресурса `try` может состоять из переменной, которая была ранее объявлена и инициализирована в программе. Но эта переменная фактически должна быть последней в программе, чтобы ей не присваивалось новое значение после инициализации.

Область применимости таких инструкций `try` ограничена ресурсами, которые реализуют интерфейс `AutoCloseable`, определенный в пакете `java.lang`. В этом интерфейсе определен метод `close()`. Интерфейс `AutoCloseable` наследуется интерфейсом `Closeable`, определенным в пакете `java.io`. Оба интерфейса реализуются классами потоков, в том числе `FileInputStream` и `FileOutputStream`. Следовательно, инструкция `try` с ресурсами может применяться вместе с потоками, включая потоки файлового ввода-вывода.

В качестве примера ниже приведена переработанная версия программы `ShowFile`, в которой инструкция `try` с ресурсами используется для автоматического закрытия файла.

```
/* В этой версии программы ShowFile инструкция try с ресурсами
   применяется для автоматического закрытия файла, когда в нем
   больше нет необходимости.
*/

import java.io.*;

class ShowFile {
   public static void main(String args[])
   {
      int i;

      // Прежде всего необходимо убедиться в том, что программе
      // передаются имена обоих файлов
      if(args.length != 1) {
         System.out.println("Использование: ShowFile имя_файла");
         return;
      }

      // Использование инструкции try с ресурсами для
      // открытия файла с последующим его закрытием после
      // того, как будет покинут блок try
      try(FileInputStream fin = new FileInputStream(args[0])) {   ◄── Блок try с ресурсами
         do {
            i = fin.read();
            if(i != -1) System.out.print((char) i);
         } while(i != -1);

      } catch(IOException exc) {
         System.out.println("Ошибка ввода-вывода: " + exc);
      }
   }
}
```

Обратите внимание на то, как открывается файл в инструкции `try` с ресурсами:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Здесь сначала объявляется переменная `fin` типа `FileInputStream`, а затем этой переменной присваивается ссылка на файл, который выступает в качестве объекта, открываемого с помощью конструктора класса `FileInputStream`. Таким образом, в данной версии программы переменная `fin` является локальной по отношению к блоку `try` и создается при входе в этот блок. При выходе из блока `try` файл, связанный с переменной `fin`, автоматически закрывается с помощью неявно вызываемого метода `close()`. Это означает, что теперь отсутствует риск того, что вы забудете закрыть файл путем явного вызова метода `close()`. В этом и состоит главное преимущество автоматического управления ресурсами.

Важно понимать, что ресурс, объявленный в инструкции `try`, неявно имеет модификатор `final`. Это означает, что после создания ресурсной переменной ее значение не может быть изменено. Кроме того, ее область действия ограничивается блоком `try`.

С помощью одной подобной инструкции `try` можно управлять несколькими ресурсами. Для этого достаточно указать список объявлений ресурсов, разделенных точкой с запятой. В качестве примера ниже приведена переработанная версия программы `CopyFile`. В этой версии оба ресурса, `fin` и `fout`, управляются одной инструкцией `try`.

```
/* Версия программы CopyFile, в которой используется инструкция
   try с ресурсами. В ней демонстрируется управление двумя
   ресурсами (в данном случае — файлами) с помощью единственной
   инструкции try.
*/

import java.io.*;

class CopyFile {
   public static void main(String args[]) throws IOException
   {
      int i;

      // Прежде всего необходимо убедиться в том, что программе
      // передаются имена обоих файлов
      if(args.length != 2) {
         System.out.println("Использование: CopyFile -
                             источник и назначение ");
         return;
      }

      // Открытие двух файлов и управление
      // ими с помощью инструкции try
```

```
      try (FileInputStream fin = new FileInputStream(args[0]);
           FileOutputStream fout = new FileOutputStream(args[1]))
      {
         do {
            i = fin.read();
            if(i != -1) fout.write(i);
         } while(i != -1);

      } catch(IOException exc) {
         System.out.println("Ошибка ввода-вывода: " + exc);
      }
   }
}
```

Управление двумя ресурсами

Обратите внимание на то, как входной и выходной файлы открываются в инструкции `try`.

```
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{
```

По завершении этого блока `try` оба файла, на которые ссылаются переменные `fin` и `fout`, будут автоматически закрыты. Если сравнить эту версию программы с предыдущей версией, то можно заметить, что ее исходный код намного компактнее. Возможность создания более компактного кода является еще одним, дополнительным преимуществом инструкции `try` с ресурсами.

Стоит упомянуть еще об одной особенности инструкции `try` с ресурсами. Вообще говоря, возникшее при выполнении блока `try` исключение может породить другое исключение при закрытии ресурса в блоке `finally`. В случае "обычной" инструкции `try` первоначальное исключение теряется, будучи прерванным вторым исключением. Но в случае инструкции `try` с ресурсами второе исключение *подавляется*. При этом оно не теряется, а просто добавляется в список подавленных исключений, связанных с первым исключением. Этот список можно получить, вызвав метод `getSuppressed()`, определенный в классе `Throwable`.

Благодаря своим преимуществам инструкция `try` с ресурсами будет использоваться во многих оставшихся примерах программ в книге. Однако не менее важным остается и умение использовать рассмотренный ранее традиционный способ освобождения ресурсов с помощью явного вызова метода `close()`. И на то имеется ряд веских причин. Во-первых, среди уже существующих и повсеместно эксплуатируемых программ на Java немало таких, в которых применяется традиционный способ управления ресурсами. Поэтому нужно как следует усвоить традиционный подход и уметь использовать его для сопровождения устаревшего кода. Во-вторых, переход к использованию версии JDK 7 или выше может произойти не сразу, а следовательно, придется работать с предыдущей версией данного комплекта. В этом случае воспользоваться преимуществами инструкции `try` с ресурсами не удастся и нужно будет применять

традиционный способ управления ресурсами. И наконец, в некоторых случаях закрытие ресурса явным образом оказывается более эффективным, чем его автоматическое освобождение. И все же, если вы работаете с современными версиями Java, вариант автоматического управления ресурсами, как более рациональный и надежный, следует считать предпочтительным.

Чтение и запись двоичных данных

В приведенных до сих пор примерах программ читались и записывались байтовые значения, содержащие символы в коде ASCII. Но аналогичным образом можно организовать чтение и запись любых типов данных. Допустим, требуется создать файл, содержащий значения типа `int`, `double` или `short`. Для чтения и записи простых типов данных в Java предусмотрены классы `DataInputStream` и `DataOutputStream`.

Класс `DataOutputStream` реализует интерфейс `DataOutput`, в котором определены методы, позволяющие записывать в файл значения любых примитивных типов. Следует, однако, иметь в виду, что данные записываются во внутреннем двоичном формате, а не в виде последовательности символов. Методы, наиболее часто применяемые для записи простых типов данных в Java, приведены в табл. 10.5. При возникновении ошибки ввода-вывода каждый из них может генерировать исключение `IOException`.

Таблица 10.5. Наиболее часто используемые методы вывода данных, определенные в классе `DataOutputStream`

Метод	Описание
`void writeBoolean(boolean val)`	Записывает логическое значение, определяемое параметром *val*
`void writeByte(int val)`	Записывает младший байт целочисленного значения, определяемого параметром *val*
`void writeChar(int val)`	Записывает значение, определяемое параметром *val*, интерпретируя его как символ
`void writeDouble(double val)`	Записывает значение типа `double`, определяемое параметром *val*
`void writeFloat(float val)`	Записывает значение типа `float`, определяемое параметром *val*
`void writeInt(int val)`	Записывает значение типа `int`, определяемое параметром *val*
`void writeLong(long val)`	Записывает значение типа `long`, определяемое параметром *val*
`void writeShort(int val)`	Записывает целочисленное значение, определяемое параметром *val*, преобразуя его в тип `short`

Ниже приведен конструктор класса `DataOutputStream`. Обратите внимание на то, что при вызове ему передается экземпляр класса `OutputStream`.

```
DataOutputStream(OutputStream outputStream)
```

Здесь *outputStream* — выходной поток, в который записываются данные. Для того чтобы организовать запись данных в файл, следует передать конструктору в качестве параметра *outputStream* объект типа `FileOutputStream`.

Класс `DataInputStream` реализует интерфейс `DataInput`, предоставляющий методы для чтения всех примитивных типов данных Java (табл. 10.6). При возникновении ошибки ввода-вывода каждый из них может генерировать исключение `IOException`. Класс `DataInputStream` построен на основе экземпляра класса `InputStream`, перекрывая его методами для чтения различных типов данных Java. Однако в потоке типа `DataInputStream` данные читаются в двоичном виде, а не в удобной для чтения форме. Ниже приведен конструктор класса `DataInputStream`:

```
DataInputStream(InputStream inputStream)
```

Здесь *inputStream* — это поток, связанный с создаваемым экземпляром класса `DataInputStream`. Для того чтобы организовать чтение данных из файла, следует передать конструктору в качестве параметра *inputStream* объект типа `FileInputStream`.

Таблица 10.6. Наиболее часто используемые методы ввода данных, определенные в классе `DataInputStream`

Метод	Описание
`boolean readBoolean()`	Читает значение типа `boolean`
`byte readByte()`	Читает значение типа `byte`
`char readChar()`	Читает значение типа `char`
`double readDouble()`	Читает значение типа `double`
`float readFloat()`	Читает значение типа `float`
`int readInt()`	Читает значение типа `int`
`long readLong()`	Читает значение типа `long`
`short readShort()`	Читает значение типа `short`

Ниже приведен пример программы, демонстрирующий использование классов `DataOutputStream` и `DataInputStream`. В этой программе данные разных типов сначала записываются в файл, а затем читаются из него.

```
// Запись и чтение двоичных данных

import java.io.*;

class RWData {
   public static void main(String args[])
```

```
    {
        int i = 10;
        double d = 1023.56;
        boolean b = true;

        // Запись ряда значений
        try (DataOutputStream dataOut =
                new DataOutputStream(new FileOutputStream("testdata")))
        {
            System.out.println("Записано: " + i);
            dataOut.writeInt(i);

            System.out.println("Записано: " + d);
            dataOut.writeDouble(d);

            System.out.println("Записано: " + b);
            dataOut.writeBoolean(b);

            System.out.println("Записано: " + 12.2 * 7.4);
            dataOut.writeDouble(12.2 * 7.4);
        }
        catch(IOException exc) {
            System.out.println("Ошибка при записи");
            return;
        }

        System.out.println();

        // А теперь прочитать записанные значения
        try (DataInputStream dataIn =
                  new DataInputStream(new FileInputStream("testdata")))
        {
            i = dataIn.readInt();
            System.out.println("Прочитано: " + i);

            d = dataIn.readDouble();
            System.out.println("Прочитано: " + d);

            b = dataIn.readBoolean();
            System.out.println("Прочитано: " + b);

            d = dataIn.readDouble();
            System.out.println("Прочитано: " + d);
        }
        catch(IOException exc) {
            System.out.println("Ошибка при чтении");
        }
    }
}
```

Запись двоичных данных

Считывание двоичных данных

В результате выполнения этой программы получим следующий результат.

```
Записано: 10
Записано: 1023.56
Записано: true
Записано: 90.28

Прочитано: 10
Прочитано: 1023.56
Прочитано: true
Прочитано: 90.28
```

Упражнение 10.1 Утилита сравнения файлов

CompFiles.java

В этом упражнении нам предстоит создать простую, но очень полезную утилиту для сравнения содержимого файлов. В ходе выполнения этой служебной программы сначала открываются два сравниваемых файла, а затем данные читаются из них и сравниваются по соответствующему количеству байтов. Если на какой-то стадии операция сравнения дает отрицательный результат, это означает, что содержимое обоих файлов не одинаково. Если же конец обоих файлов достигается одновременно, это означает, что они содержат одинаковые данные. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл CompFiles.java.
2. Введите в файл CompFiles.java приведенный ниже исходный код.

```
/*
   Упражнение 10.1

   Сравнение двух файлов.

   При вызове этой программы следует указать имена
   сравниваемых файлов. Например, чтобы сравнить файл
   FIRST.TXT с файлом SECOND.TXT, в командной строке
   нужно ввести следующую команду:

   java CompFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CompFiles {
   public static void main(String args[])
   {
      int i=0, j=0;

      // Прежде всего необходимо убедиться в том, что программе
      // передаются имена обоих файлов
      if(args.length !=2 ) {
```

```
            System.out.println("Использование: CompFiles файл1 файл2");
            return;
        }

        // Сравнение файлов
        try (FileInputStream f1 = new FileInputStream(args[0]);
             FileInputStream f2 = new FileInputStream(args[1]))
        {
            // Проверка содержимого каждого файла
            do {
                i = f1.read();
                j = f2.read();
                if(i != j) break;
            } while(i != -1 && j != -1);

            if(i != j)
                System.out.println("Содержимое файлов отличается");
            else
                System.out.println("Содержимое файлов совпадает");
        } catch(IOException exc) {
            System.out.println("Ошибка ввода-вывода: " + exc);
        }
    }
}
```

3. Перед запуском программы скопируйте файл `CompFiles.java` во временный файл `temp`, а затем введите в командной строке такую команду:

   ```
   java CompFiles CompFiles.java temp
   ```

   Программа сообщит, что файлы имеют одинаковое содержимое. Далее сравните файл `CompFiles.java` с рассмотренным ранее файлом `CopyFile.java`, введя в командной строке следующую команду:

   ```
   java CompFiles CompFiles.java CopyFile.java
   ```

   Эти файлы имеют различное содержимое, о чем и сообщит программа `CompFiles`.
4. Попытайтесь самостоятельно включить в программу `CompFiles` дополнительные возможности. В частности, предусмотрите возможность выполнять сравнение без учета регистра символов. Кроме того, программу `CompFiles` можно доработать так, чтобы она выводила номер позиции, в которой находится первая пара отличающихся символов.

Файлы с произвольным доступом

До сих пор мы имели дело с последовательными файлами, содержимое которых вводилось и выводилось побайтово, т.е. строго по порядку. Но в Java предоставляется также возможность обращаться к хранящимся в файле данным в произвольном порядке. Для этой цели предусмотрен класс `RandomAccessFile`,

инкапсулирующий файл с произвольным доступом. Класс RandomAccessFile не является производным от класса InputStream или OutputStream. Вместо этого он реализует интерфейсы DataInput и DataOutput, в которых объявлены основные методы ввода-вывода. Кроме того, он поддерживает запросы с позиционированием, т.е. позволяет задавать положение указателя файла произвольным образом. Ниже приведен конструктор класса RandomAccessFile, который мы будем использовать далее.

```
RandomAccessFile(String имя_файла, String доступ)
      throws FileNotFoundException
```

Здесь конкретный файл указывается с помощью параметра *имя_файла*, а параметр *доступ* определяет, какой именно тип доступа будет использоваться для обращения к файлу. Если параметр *доступ* принимает значение "r", то данные могут читаться из файла, но не записываться в него. Если же указан тип доступа "rw", то файл открывается как для чтения, так и для записи. Параметр *доступ* также может принимать значения "rws" и "rwd" (для локальных устройств), которые определяют немедленное сохранение файла на физическом устройстве.

Метод seek(), общая форма объявления которого приведена ниже, предназначен для установки текущего положения указателя файла.

```
void seek(long новая_позиция) throws IOException
```

Здесь параметр *новая_позиция* определяет новое положение указателя файла в байтах относительно начала файла. Операция чтения или записи, следующая после вызова метода seek(), будет выполняться относительно нового положения указателя.

В классе RandomAccessFile определены методы read() и write(). Этот класс реализует также интерфейсы DataInput и DataOuput, т.е. в нем доступны методы чтения и записи простых типов, например readInt() и writeDouble().

Ниже приведен пример программы, демонстрирующий ввод-вывод с произвольным доступом. В этой программе шесть значений типа double сначала записываются в файл, а затем читаются из него, причем порядок их чтения отличается от порядка записи.

```
// Демонстрация произвольного доступа к файлам

import java.io.*;

class RandomAccessDemo {
   public static void main(String args[])
   {
      double data[] = { 19.4, 10.1, 123.54, 33.0, 87.9, 74.25 };
      double d;

      // Открыть и использовать файл с произвольным доступом
      try (RandomAccessFile raf =
         new RandomAccessFile("random.dat", "rw")) ◄──── Открыть файл с
                                                          произвольным доступом
```

```
        {
           // Запись значения в файл
           for(int i=0; i < data.length; i++) {
              raf.writeDouble(data[i]);
           }

           // Считывание отдельных значений из файла
           raf.seek(0); // найти первое значение типа double
           d = raf.readDouble();
           System.out.println("Первое значение: " + d);

           raf.seek(8) ; // найти второе значение типа double
           d = raf.readDouble();
           System.out.println("Второе значение: " + d);

           raf.seek(8 * 3); // найти четвертое значение типа double
           d = raf.readDouble();
           System.out.println("Четвертое значение: " + d);

           System.out.println();

           // Прочитать значения через одно
           System.out.println("Чтение значений с нечетными
                              порядковыми номерами: ");
           for(int i=0; i < data.length; i+=2) {
              raf.seek(8 * i); // найти i-е значение типа double
              d = raf.readDouble();
              System.out.print(d + " ");
           }
        }
        catch(IOException exc) {
           System.out.println("Ошибка ввода-вывода: " + exc);
        }
    }
}
```

Установка указателя на файл с помощью метода `seek()`

Результат выполнения данной программы выглядит следующим образом.

```
Первое значение: 19.4
Первое значение: 10.1
Четвертое значение 33.0

Чтение значений с нечетными порядковыми номерами:
19.4 123.54 87.9
```

Отдельное замечание следует сделать относительно позиций расположения значений в файле. Поскольку для хранения значения типа `double` требуется 8 байтов, каждое последующее значение начинается на 8 байтовой границе предыдущего значения. Иными словами, первое числовое значение начинается с нулевого байта, второе — с 8-го байта, третье — с 16-го и т.д. Поэтому для чтения четвертого значения указатель файла должен быть установлен при вызове метода `seek()` на позиции 24-го байта.

Спросим у эксперта

ВОПРОС. В документации по JDK упоминается класс `Console`. Можно ли воспользоваться им для обмена данными с консолью?

ОТВЕТ. Да, можно. Класс `Console` впервые появился в версии Java 6 и служит для ввода-вывода данных на консоль. Этот класс создан в основном для удобства работы с консолью, поскольку бо́льшая часть его функциональных возможностей доступна в стандартных потоках ввода-вывода `System.in` и `System.out`. Но пренебрегать классом `Console` все же не следует, ведь с его помощью можно упростить некоторые типы операций с консолью, в том числе чтение символьных строк, вводимых с клавиатуры.

Класс `Console` не предоставляет конструкторы. Для получения объекта данного типа следует вызывать статический метод `System.console()`, который также был включен в версию Java 6. Ниже приведена общая форма объявления этого метода:

```
static Console console()
```

Если консоль доступна, то этот метод возвращает ссылку на соответствующий объект. В противном случае возвращается пустое значение `null`. Консоль доступна не всегда: обращение к ней запрещено, если программа выполняется в фоновом режиме.

В классе `Console` определен ряд методов, поддерживающих ввод-вывод, например `readLine()` и `printf()`. В нем также содержится метод `readPassword()`, предназначенный для получения пароля. При вводе пароля с клавиатуры его символы отображаются на экране с помощью замещающих знаков, не раскрывая пароль. С помощью средств класса `Console` можно также получить ссылки на объекты типа `Reader` и `Writer`, связанные с консолью. Таким образом, класс `Console` может оказаться очень полезным при написании некоторых видов приложений.

Использование символьных потоков Java

Как говорилось в предыдущих разделах, байтовые потоки Java отличаются эффективностью и удобством использования. Но во всем, что касается ввода-вывода символов, они далеки от идеала. Для преодоления этого недостатка в Java определены классы символьных потоков. На вершине иерархии классов, поддерживающих символьные потоки, находятся абстрактные классы `Reader` и `Writer`. Методы класса `Reader` приведены в табл. 10.7, а методы класса `Writer` — в табл. 10.8. В большинстве этих методов может генерироваться исключение `IOException`. Методы, определенные в этих абстрактных классах, доступны во всех их подклассах. В совокупности эти методы предоставляют минимальный набор функций ввода-вывода, которые будут иметь все символьные потоки.

Таблица 10.7. Методы, определенные в классе `Reader`

Метод	Описание
`abstract void close()`	Закрывает источник ввода. Дальнейшие попытки чтения будут генерировать исключение `IOException`
`void mark (int` *`numChars`*`)`	Помещает в текущую позицию входного потока метку, которая будет находиться там до тех пор, пока не будет прочитано количество байтов, определяемое параметром *`numChars`*
`boolean markSupported()`	Возвращает значение `true`, если методы `mark()` и `reset()` поддерживаются вызывающим потоком
`int read()`	Возвращает целочисленное представление следующего символа в вызывающем входном потоке. По достижении конца потока возвращается значение –1
`int read(char` *`buffer`*`[])`	Пытается прочитать *`buffer.length`* символов в массив *`buffer`*, возвращая фактическое количество успешно прочитанных символов. По достижении конца потока возвращается значение –1
`abstract int read(char` *`buffer`*`[], int` *`offset`*`, int` *`numChars`*`)`	Пытается прочитать количество символов, определяемое параметром *`numChars`*, в массив *`buffer`*, начиная с элемента *`buffer`*`[`*`offset`*`]`. По достижении конца потока возвращается значение –1
`int read(CharBuffer` *`buffer`*`)`	Пытается заполнить буфер, определяемый параметром *`buffer`*, и возвращает количество успешно прочитанных символов. По достижении конца потока возвращается значение –1. `CharBuffer` — это класс, инкапсулирующий последовательность символов, например строку
`boolean ready()`	Возвращает значение `true`, если следующий запрос на получение символа может быть выполнен без ожидания. В противном случае возвращается значение `false`
`void reset()`	Сбрасывает входной указатель на ранее установленную метку
`long skip(long` *`numChars`*`)`	Пропускает *`numChars`* символов во входном потоке, возвращая фактическое количество пропущенных символов

Таблица 10.8. Методы, определенные в классе `Writer`

Метод	Описание
`Writer append(char ch)`	Добавляет символ *ch* в конец вызывающего выходного потока, возвращая ссылку на вызывающий поток
`Writer append(CharSequence chars)`	Добавляет последовательность символов *chars* в конец вызывающего потока, возвращая ссылку на вызывающий поток. `CharSequence` — это интерфейс, определяющий операции над последовательностями символов, выполняемые в режиме "только чтение"
`Writer append(CharSequence chars, int begin, int end)`	Добавляет последовательность символов *chars* в конец текущего потока, начиная с позиции, определяемой параметром *begin*, и заканчивая позицией, определяемой параметром *end*. Возвращает ссылку на вызывающий поток. `CharSequence` — это интерфейс, определяющий операции над последовательностями символов, выполняемые в режиме "только чтение"
`abstract void close()`	Закрывает выходной поток. Дальнейшие попытки чтения будут генерировать исключение `IOException`
`abstract void flush()`	Выполняет принудительную передачу содержимого выходного буфера в место назначения (тем самым очищая выходной буфер)
`void write(int ch)`	Записывает один символ в вызывающий выходной поток. Обратите внимание на то, что параметр имеет тип `int`, что позволяет вызывать метод `write()` с выражениями, не приводя их к типу `char`
`void write(char buffer[])`	Записывает полный массив символов *buffer* в вызывающий выходной поток
`abstract void write(char buffer[], int offset, int numChars)`	Записывает часть массива символов *buffer* в количестве *numChars* символов, начиная с элемента *buffer*[*offset*], в вызывающий выходной поток
`void write(String str)`	Записывает строку *str* в вызывающий выходной поток
`void write(String str, int offset, int numChars)`	Записывает часть строки *str* в количестве *numChars* символов, начиная с позиции, определяемой параметром *offset*, в вызывающий поток

Консольный ввод с использованием символьных потоков

В случае программ, подлежащих интернационализации, для ввода символов с клавиатуры проще и удобнее использовать символьные потоки, а не байтовые. Но поскольку `System.in` — это байтовый поток, для него придется построить оболочку в виде класса, производного от класса `Reader`. Наиболее подходящим для ввода с консоли является класс `BufferedReader`, поддерживающий буферизованный входной поток. Однако объект типа `BufferedReader` нельзя создать непосредственно на основе стандартного потока ввода `System.in`. Сначала нужно преобразовать байтовый поток в символьный. Для этого используется класс `InputStreamReader`, преобразующий байты в символы. Чтобы получить объект типа `InputStreamReader`, связанный с потоком стандартного ввода `System.in`, необходимо воспользоваться следующим конструктором:

```
InputStreamReader(InputStream inputStream)
```

Поток ввода `System.in` — это экземпляр класса `InputStream`, и поэтому его можно указать в качестве параметра *inputStream* данного конструктора.

Затем с помощью объекта, созданного на основе объекта типа `InputStreamReader`, можно получить объект типа `BufferedReader`, используя следующий конструктор:

```
BufferedReader(Reader inputReader)
```

где *inputReader* — поток, который связывается с создаваемым экземпляром класса `BufferedReader`. Объединяя обращения к указанным выше конструкторам в одну операцию, мы получаем приведенную ниже строку кода. В ней создается объект типа `BufferedReader`, связанный с клавиатурой.

```
BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
```

После выполнения этого кода переменная `br` будет содержать ссылку на символьный поток, связанный с консолью через поток ввода `System.in`.

Чтение символов

Чтение символов из потока `System.in` с помощью метода `read()` осуществляется в основном так, как если бы это делалось с помощью байтовых потоков. Ниже приведены общие формы объявления трех версий метода `read()`, предусмотренных в классе `BufferedReader`.

```
int read() throws IOException
int read(char data[]) throws IOException
int read(char data[], int start, int max) throws IOException
```

Первая версия метода `read()` читает одиночный символ в кодировке Unicode. По достижении конца потока метод возвращает значение –1. Вторая версия метода `read()` читает символы из входного потока и помещает их в массив. Этот процесс продолжается до тех пор, пока не будет достигнут конец потока, или пока массив `data` не заполнится символами, или не возникнет ошибка, в зависимости от того, какое из этих событий произойдет первым.

В этом случае метод возвращает число прочитанных символов, а в случае достижения конца потока — значение –1. Третья версия метода read() помещает прочитанные символы в массив data, начиная с элемента, определяемого параметром start. Максимальное число символов, которые могут быть записаны в массив, определяется параметром max. В данном случае метод возвращает число прочитанных символов или значение –1, если достигнут конец потока. При возникновении ошибки в каждой из вышеперечисленных версий метода read() генерируется исключение IOException. При чтении данных из потока ввода System.in конец потока устанавливается нажатием клавиши <Enter>.

Ниже приведен пример программы, демонстрирующий применение метода read() для чтения символов с консоли. Символы читаются до тех пор, пока пользователь не введет точку. Следует иметь в виду, что исключения, которые могут быть сгенерированы при выполнении данной программы, обрабатываются за пределами метода main(). Как уже отмечалось, такой подход к обработке ошибок является типичным при чтении данных с консоли. По желанию можно использовать другой механизм обработки ошибок.

```
// Использование класса BufferedReader
// для чтения символов с консоли
import java.io.*;

class ReadChars {
   public static void main(String args[]) throws IOException
   {
      char c;
      BufferedReader br = new
         BufferedReader(new InputStreamReader(System.in));

      System.out.println("Введите символы; окончание ввода -
                         символ точки");

      // считывание символов
      do {
         c = (char) br.read();
         System.out.println(c);
      }  while(c != '.');
   }
}
```

Создание класса BufferedRead, связанного с потоком System.In

Результат выполнения данной программы выглядит следующим образом.

```
Введите символы; окончание ввода - символ точки
One Two.
O
n
e
T
w
o
.
```

Чтение строк

Для ввода строки с клавиатуры используют метод `readLine()` класса `BufferedReader`. Вот общая форма объявления этого метода:

```
String readLine() throws IOException
```

Этот метод возвращает объект типа `String`, содержащий прочитанные символы. При попытке прочитать строку по достижении конца потока метод возвращает значение `null`.

Ниже приведен пример программы, демонстрирующий использование класса `BufferedReader` и метода `readLine()`. В этой программе текстовые строки читаются и отображаются до тех пор, пока не будет введено слово "`stop`".

```
// Чтение символьных строк с консоли с использованием
// класса BufferedReader
import java.io.*;

class ReadLines {
   public static void main(String args[]) throws IOException
   {
      // Создать объект типа BufferedReader,
      // связанный с потоком System.in
      BufferedReader br = new BufferedReader(new
                              InputStreamReader(System.in));

      String str;

      System.out.println("Введите текстовые строки");
      System.out.println("Признак конца ввода - строка 'stop' ");
      do {
         str = br.readLine(); ◄——— Использование метода readLine() из класса
                                   BufferedRead для чтения строки текста
         System.out.println(str);
      } while(!str.equals("stop"));
   }
}
```

Вывод на консоль с использованием символьных потоков

Несмотря на то что поток стандартного вывода `System.out` вполне пригоден для вывода на консоль, в большинстве случаев такой подход рекомендуется использовать лишь в целях отладки или при создании очень простых программ наподобие тех, которые приведены в данной книге в качестве примеров. В реальных прикладных программах на Java вывод на консоль обычно организуется через поток `PrintWriter`. Класс `PrintWriter` является одним из классов, представляющих символьные потоки. Как уже упоминалось, применение потоков упрощает локализацию прикладных программ.

В классе `PrintWriter` определен целый ряд конструкторов. Далее будет использоваться следующий конструктор:

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

Здесь в качестве первого параметра, *outputStream*, конструктору передается объект типа OutputStream, а второй параметр, *flushOnNewline*, указывает, должен ли буфер выходного потока сбрасываться каждый раз, когда вызывается (среди прочих других) метод println(). Если параметр *flushOnNewline* имеет значение true, сбрасывание буфера выполняется автоматически.

В классе PrintWriter поддерживаются методы print() и println() для всех типов, включая Object. Следовательно, методы print() и println() можно использовать точно так же, как и совместно с потоком вывода System.out. Если значение аргумента не относится к простому типу, то методы класса PrintWriter вызывают метод toString() для объекта, указанного в качестве параметра, а затем выводят результат.

Для вывода данных на консоль через поток типа PrintWriter следует указать System.out в качестве выходного потока и обеспечить вывод данных из буфера после каждого вызова метода println(). Например, при выполнении следующей строки кода создается объект типа PrintWriter, связанный с консолью:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

Ниже приведен пример программы, демонстрирующий использование класса PrintWriter для организации вывода на консоль.

```
// Использование класса PrintWriter
import java.io.*;

public class PrintWriterDemo {
   public static void main(String args[]) {
      PrintWriter pw = new PrintWriter(System.out, true);
      int i = 10;
      double d = 123.65;

      pw.println("Использование класса PrintWriter");
      pw.println(i);
      pw.println(d);

      pw.println(i + " + " + d + " = " + (i+d));
   }
}
```

Создание класса PrintWriter, связанного с потоком System.Out

Выполнение этой программы дает следующий результат.

```
Использование класса PrintWriter
10
123.65
10 + 123.65 = 133.65
```

Как ни удобны символьные потоки, не следует забывать, что для изучения языка Java или отладки программ вам будет вполне достаточно использовать поток System.out. Если вы используете поток PrintWriter, программу будет проще интернационализировать. Для небольших программ наподобие тех,

которые представлены в данной книге в виде примеров, использование потока `PrintWriter` не дает никаких существенных преимуществ по сравнению с потоком `System.out`, поэтому далее для вывода на консоль будет использоваться поток `System.out`.

Файловый ввод-вывод с использованием символьных потоков

Несмотря на то что файловые операции ввода-вывода чаще всего выполняются с помощью байтовых потоков, для этой цели можно использовать также символьные потоки. Преимущество символьных потоков заключается в том, что они оперируют непосредственно символами в кодировке Unicode. Так, если вам нужно сохранить текст в кодировке Unicode, то для этой цели лучше всего воспользоваться символьными потоками. Как правило, для файлового ввода-вывода символов используются классы `FileReader` и `FileWriter`.

Класс FileWriter

Класс `FileWriter` создает объект типа `Writer`, который можно использовать для записи данных в файл. Ниже приведены общие формы объявления двух наиболее часто используемых конструкторов данного класса.

```
FileWriter(String имя_файла) throws IOException
FileWriter(String имя_файла, boolean append) throws IOException
```

Здесь *имя_файла* обозначает полный путь к файлу. Если параметр *append* имеет значение `true`, данные записываются в конец файла, в противном случае запись осуществляется поверх существующих данных. При возникновении ошибки в каждом из указанных конструкторов генерируется исключение `IOException`. Класс `FileWriter` является производным от классов `OutputStreamWriter` и `Writer`. Следовательно, в нем доступны методы, объявленные в его суперклассах.

Ниже приведен пример небольшой программы, демонстрирующий ввод текстовых строк с клавиатуры и последующую их запись в файл `test.txt`. Набираемый текст читается до тех пор, пока пользователь не введет слово `"stop"`. Для вывода текстовых строк в файл используется класс `FileWriter`.

```
// Пример простой утилиты для ввода данных с клавиатуры и
// записи их на диск, демонстрирующий использование класса
// FileWriter

import java.io.*;

class KtoD {
   public static void main(String args[])
   {
```

```
      String str;
      BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
      System.out.println("Признак конца ввода - строка 'stop' ");

      try (FileWriter fw = new FileWriter("test.txt"))  ◄── Создание класса
      {                                                      FileWriter
         do {
            System.out.print(": ");
            str = br.readLine();

            if(str.compareTo("stop") == 0) break;

            str = str + "\r\n"; // добавить символы
                                // перевода строки
            fw.write(str); ◄──────────────────────── Запись строк в файл
         } while(str.compareTo("stop") != 0);
      } catch(IOException exc) {
         System.out.println("Ошибка ввода-вывода: " + exc);
      }
   }
}
```

Класс FileReader

Класс `FileReader` создает объект типа `Reader`, который можно использовать для чтения содержимого файла. Чаще всего используется следующая форма конструктора этого класса:

```
FileReader(String имя_файла) throws FileNotFoundException
```

где *имя_файла* обозначает полный путь к файлу. Если указанного файла не существует, генерируется исключение `FileNotFoundException`. Класс `FileReader` является производным от классов `InputStreamReader` и `Reader`. Следовательно, в нем доступны методы, объявленные в его суперклассах.

В приведенном ниже примере создается простая утилита, отображающая на экране содержимое текстового файла `test.txt`. Она является своего рода дополнением к утилите, рассмотренной в предыдущем разделе.

```
// Пример простой утилиты для чтения данных с диска и вывода их
// на экран, демонстрирующий использование класса FileReader

import java.io.*;

class DtoS {
   public static void main(String args[]) {
      String s;

      // Создать и использовать объект FileReader, помещенный
      // в оболочку на основе класса BufferedReader
      try (BufferedReader br =              Создание класса FileReader ─┐
         new BufferedReader(new FileReader("test.txt"))) ◄──────────────┘
```

```
        {
           while((s = br.readLine()) != null) {
              System.out.println(s);
           }
        } catch(IOException exc) {
           System.out.println("Ошибка ввода-вывода: " + exc);
        }
    }
}
```

Обратите внимание на то, что для потока `FileReader` создается оболочка на основе класса `BufferedReader`. Благодаря этому появляется возможность обращаться к методу `readLine()`. Кроме того, закрытие потока типа `BufferedReader`, на который в данном примере ссылается переменная `br`, автоматически приводит к закрытию файла.

Спросим у эксперта

ВОПРОС. Я слышал об отдельном пакете ввода-вывода, называемом NIO, который включает классы для поддержки ввода-вывода. Что он собой представляет?

ОТВЕТ. Пакет NIO (сокращение от *New I/O*) был реализован в версии JDK 1.4. В нем поддерживается канальный подход к организации операций ввода-вывода. Классы новой подсистемы ввода-вывода относятся к пакету `java.nio` и его подчиненным пакетам, включая `java.nio.channels` и `java.nio.charset`.

Пакет NIO опирается на два основных понятия: *буфер* и *канал*. Буфер содержит данные, а канал представляет собой соединение, устанавливаемое с устройством ввода-вывода, например с файлом на диске или сетевым сокетом. Как правило, для применения новой подсистемы ввода-вывода нужно получить канал доступа к устройству и буфер для хранения данных. После этого можно выполнять необходимые операции с буфером, в частности, вводить или выводить данные.

Кроме буфера и канала, в NIO используются также понятия набора символов и селектора. *Набор символов* определяет способ преобразования байтов в символы. Для представления последовательности символов в виде набора байтов используется *шифратор*. Обратное преобразование осуществляет *дешифратор*. А *селектор* поддерживает неблокирующий мультиплексный ввод-вывод с ключом шифрования. Иными словами, селекторы позволяют выполнять обмен данными по нескольким каналам. Селекторы находят наибольшее применение в каналах, поддерживаемых сетевыми сокетами.

В версии JDK 7 новая подсистема ввода-вывода была значительно усовершенствована, и поэтому нередко ее называют *NIO.2*. К числу ее усовершенствований относятся три новых пакета (`java.nio.file`, `java.nio.file.`

attribute и java.nio.file.spi), ряд новых классов, интерфейсов и методов, а также непосредственная поддержка потокового ввода-вывода. Все эти дополнения в значительной степени расширили область применения NIO, и особенно это касается обработки файлов.

Однако новая подсистема ввода-вывода не призвана заменить классы ввода-вывода, существующие в пакете java.io. Классы NIO лишь дополняют стандартную подсистему ввода-вывода, предлагая альтернативный подход, вполне уместный в ряде случаев.

Использование классов-оболочек для преобразования числовых строк

Прежде чем завершить обсуждение средств ввода-вывода, необходимо рассмотреть еще один прием, который оказывается весьма полезным при чтении числовых строк. Как вам уже известно, метод println() предоставляет удобные средства для вывода на консоль различных типов данных, включая целые числа и числа с плавающей точкой. Он автоматически преобразует числовые значения в удобную для чтения форму. Но в Java отсутствует метод, который читал бы числовые строки и преобразовывал их во внутреннюю двоичную форму. Например, не существует варианта метода read(), который читал бы числовую строку "100" и автоматически преобразовывал ее в целое число, пригодное для хранения в переменной типа int. Но для этой цели в Java имеются другие средства. И проще всего подобное преобразование осуществляется с помощью так называемых *оболочек типов* (объектных оболочек) Java.

Объектные оболочки в Java представляют собой классы, которые инкапсулируют простые типы. Оболочки типов необходимы, поскольку простые типы не являются объектами, что ограничивает их применение. Так, простой тип нельзя передать методу по ссылке. Для того чтобы исключить ненужные ограничения, в Java были предусмотрены классы, соответствующие каждому из простых типов.

Объектными оболочками являются классы Double, Float, Long, Integer, Short, Byte, Character и Boolean, которые предоставляют обширный ряд методов, позволяющих полностью интегрировать простые типы в иерархию объектов Java. Кроме того, в классах-оболочках числовых типов содержатся методы, предназначенные для преобразования числовых строк в соответствующие двоичные эквиваленты. Эти методы приведены ниже. Каждый из них возвращает двоичное значение, соответствующее числовой строке.

Оболочка типа	Метод преобразования
Double	static double parseDouble(String *str*) throws NumberFormatException
Float	static float parseFloat(String *str*) throws NumberFormatException
Long	static long parseLong(String *str*) throws NumberFormatException
Integer	static int parseInt(String *str*) throws NumberFormatException
Short	static short parseShort(String *str*) throws NumberFormatException
Byte	static byte parseByte(String *str*) throws NumberFormatException

Оболочки целочисленных типов также предоставляют дополнительный метод синтаксического анализа, позволяющий задавать основание системы счисления.

Методы синтаксического анализа позволяют без труда преобразовать во внутренний формат числовые значения, введенные в виде символьных строк с клавиатуры или из текстового файла. Ниже приведен пример программы, демонстрирующий применение для этих целей методов `parseInt()` и `parseDouble()`. Эта программа вычисляет среднее арифметическое ряда чисел, введенных пользователем с клавиатуры. Сначала пользователю предлагается указать количество подлежащих обработке числовых значений, а затем программа вводит числа с клавиатуры, используя метод `readLine()`, и с помощью метода `parseInt()` преобразует символьную строку в целочисленное значение. Далее осуществляется ввод числовых значений и последующее их преобразование в тип `double` с помощью метода `parseDouble()`.

```
/* Данная программа находит среднее арифметическое для
   ряда чисел, введенных пользователем с клавиатуры. */

import java.io.*;

class AvgNums {
   public static void main(String args[]) throws IOException
   {
      // Создание объекта типа BufferedReader,
      // использующего поток ввода System.in
      BufferedReader br = new
         BufferedReader(new InputStreamReader(System.in));
      String str;
      int n;
      double sum = 0.0;
      double avg, t;

      System.out.print("Сколько чисел нужно ввести: ");
      str = br.readLine();
```

```
        try {
            n = Integer.parseInt(str);  ◄—— Преобразование строки в тип int
        }
        catch(NumberFormatException exc) {
            System.out.println("Неверный формат");
            n = 0;
        }

        System.out.println("Ввод " + n + " значений");
        for(int i=0; i < n ; i++) {
            System.out.print(": ");
            str = br.readLine();
            try {
                t = Double.parseDouble(str);  ◄—— Преобразование строки
            } catch(NumberFormatException exc) {    в тип double
                System.out.println("Неверный формат");
                t = 0.0;
            }
            sum += t;
        }
        avg = sum / n;
        System.out.println("Среднее значение: " + avg);
    }
}
```

Выполнение этой программы может дать, например, следующий результат.

```
Сколько чисел нужно ввести: 5
Ввод 5 значений
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
Среднее значение: 3.3
```

Спросим у эксперта

ВОПРОС. Могут ли объектные оболочки простых типов выполнять другие функции, кроме описанных в этом разделе?

ОТВЕТ. Классы оболочек простых типов предоставляют ряд методов, помогающих интегрировать эти типы данных в иерархию объектов. Например, различные механизмы хранения данных, предусмотренные в библиотеке Java, включая отображения, списки и множества, взаимодействуют только с объектами. Поэтому для сохранения целочисленного значения в списке его следует сначала преобразовать в объект. Оболочки типов предоставляют также метод `compareTo()` для сравнения текущего значения с заданным, метод `equals()` для проверки равенства двух объектов, а также методы, возвращающие значение объекта в разных формах записи. К оболочкам типов мы еще вернемся в главе 12, когда речь пойдет об автоупаковке.

Упражнение 10.2 Создание справочной системы, находящейся на диске

FileHelp.javajava

В упражнении 4.1 был создан класс `Help`, позволяющий отображать сведения об инструкциях Java. Справочная информация хранилась в самом классе, а пользователь выбирал требуемые сведения из меню. И хотя такая справочная система выполняет свои функции, подход к ее разработке был выбран далеко не самый лучший. Например, если потребуется добавить или изменить какие-либо сведения, вам придется внести изменения в исходный код программы, которая реализует справочную систему. Кроме того, выбирать пункт меню по его номеру не очень удобно, а если количество пунктов велико, то такой способ вообще непригоден. В этом упражнении нам предстоит устранить недостатки, имеющиеся в справочной системе, расположив справочную информацию на диске.

В новом варианте справочная информация должна храниться в файле. Это будет обычный текстовый файл, который можно изменять, не затрагивая исходный код программы. Для того чтобы получить справку по конкретному вопросу, следует ввести название темы. Система будет искать соответствующий раздел в файле. Если поиск завершится успешно, справочная информация будет выведена на экран. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл, в котором хранится справочная информация и который будет использоваться в справочной системе. Это должен быть обычный текстовый файл, организованный, как показано ниже.

```
#название_темы_1
Информация по теме

#название_темы_2
Информация по теме

...

#название_темы_N
Информация по теме
```

Название каждой темы располагается в отдельной строке и предваряется символом `#`. Наличие специального символа в строке (в данном случае — `#`) позволяет программе быстро найти начало раздела. Под названием темы может располагаться любая справочная информация. После окончания одного раздела и перед началом другого должна быть введена пустая строка. Кроме того, в конце строк не должно быть лишних пробелов.

Ниже приведен пример простого файла со справочной информацией, который можно использовать вместе с новой версией справочной системы. В нем хранятся сведения об инструкциях Java.

```
#if
if(условие) инструкция;
else инструкция;

#switch
switch(выражение) {
   case константа:
      последовательность инструкций
      break;
      // ...
   }

#for
for(инициализация; условие; итерация) инструкция;

#while
while(условие) инструкция;

#do
do {
   инструкция;
} while (условие);

#break
break; или break метка;

#continue
continue; или continue метка;
```

Присвойте этому файлу имя helpfile.txt.

2. Создайте файл FileHelp.java.
3. Начните создание новой версии класса Help со следующих строк кода.

```
class Help {
   String helpfile; // имя файла справки

   Help(String fname) {
      helpfile = fname;
   }
```

Имя файла со справочной информацией передается конструктору класса Help и запоминается в переменной экземпляра helpfile. А поскольку каждый экземпляр класса Help содержит отдельную копию переменной helpfile, то каждый из них может взаимодействовать с отдельным файлом. Это дает возможность создавать отельные наборы справочных файлов на разные темы.

4. Добавьте в класс Help метод helpon(), код которого приведен ниже. Этот метод извлекает справочную информацию по заданной теме.

```
// Отображение справочной информации по указанной теме
boolean helpon(String what) {
```

```
   int ch;
   String topic, info;

   // Открыть справочный файл
   try (BufferedReader helpRdr =
           new BufferedReader(new FileReader(helpfile)))
   {
      do {
         // Читать символы до тех пор, пока не встретится символ #
         ch = helpRdr.read();

         // Проверить, совпадают ли темы
         if(ch == '#') {
            topic = helpRdr.readLine();
            if(what.compareTo(topic) == 0) { // найти тему
               do {
                  info = helpRdr.readLine();
                  if(info != null) System.out.println(info);
               } while((info != null) && (info.compareTo("") != 0));
                  return true;
            }
         }
      } while(ch != -1);
   }
   catch(IOException exc) {
      System.out.println("Ошибка при попытке доступа к
                         файлу справки");
      return false;
   }
   return false; // тема не найдена
}
```

Прежде всего обратите внимание на то, что в методе `helpon()` обрабатываются все исключения, связанные с вводом-выводом, поэтому в заголовке метода не указано ключевое слово `throws`. Благодаря такому подходу упрощается разработка методов, в которых используется метод `helpon()`. В вызывающем методе достаточно обратиться к методу `helpon()`, не заключая его вызов в блок `try/catch`.

Для открытия файла со справочной информацией предназначен класс `FileReader`, оболочкой которого является класс `BufferedReader`. В справочном файле содержится текст, и поэтому справочную систему удобнее локализовать через символьные потоки ввода-вывода.

Метод `helpon()` действует следующим образом. Символьная строка, содержащая название темы, передается методу в качестве параметра. Сначала метод открывает файл со справочной информацией. Затем в файле осуществляется поиск, т.е. проверяется совпадение содержимого переменной `what` и названия темы. Напомним, что в файле заголовок темы предваряется символом `#`, поэтому метод сначала ищет данный символ. Если символ

найден, следующее за ним название темы сравнивается с содержимым переменной what. Если сравниваемые строки совпадают, то отображается справочная информация по данной теме. И если заголовок темы найден, то метод helpon() возвращает логическое значение true, в противном случае — логическое значение false.

5. В классе Help содержится также метод getSelection(), который предлагает указать тему и возвращает строку, введенную пользователем.

```
// Получение темы справки
String getSelection() {
   String topic = "";

   BufferedReader br = new BufferedReader(
                           new InputStreamReader(System.in));

   System.out.print("Укажите тему: ");
   try {
      topic = br.readLine();
   }
   catch(IOException exc) {
      System.out.println("Ошибка при чтении с консоли");
   }
   return topic;
}
```

   В теле этого метода сначала создается объект типа BufferedReader, который связывается с потоком вывода System.in. Затем в нем запрашивается название темы, которое принимается и далее возвращается вызывающей части программы.

6. Ниже приведен весь исходный код программы, реализующей справочную систему на диске.

```
/*
   Упражнение 10.2

   Справочная система, использующая дисковый файл
   для хранения информации
*/

import java.io.*;

/* В классе Help открывается файл со справочной информацией,
   выполняется поиск указанной темы, а затем отображается
   справочная информация. Обратите внимание на то, что данный
   класс обрабатывает все исключения, освобождая от этого
   вызывающий код. */
class Help {
   String helpfile; // имя справочного файла

   Help(String fname) {
      helpfile = fname;
   }
```

```
    // Отображение справочной информации по указанной теме
    boolean helpon(String what) {
       int ch;
       String topic, info;

       // Открыть справочный файл
       try (BufferedReader helpRdr =
               new BufferedReader(new FileReader(helpfile)))
       {
          do {
             // Читать символы до тех пор,
             // пока не встретится символ #
             ch = helpRdr.read();

             // Проверить, совпадают ли темы
             if(ch == '#') {
                topic = helpRdr.readLine();
                if(what.compareTo(topic) == 0) { // найти тему
                   do {
                      info = helpRdr.readLine();
                      if(info != null) System.out.println(info);
                   } while((info != null) &&
                           (info.compareTo("") != 0));
                   return true;
                }
             }
          } while(ch != -1);
       }
       catch(IOException exc) {
          System.out.println("Ошибка при попытке доступа
                             к файлу справки");
          return false;
       }
       return false; // тема не найдена
    }

    // Получение темы справки
    String getSelection() {
       String topic = "";

       BufferedReader br = new BufferedReader(
                             new InputStreamReader(System.in));

       System.out.print("Укажите тему: ");
       try {
          topic = br.readLine();
       }
       catch(IOException exc) {
          System.out.println("Ошибка при чтении с консоли");
       }
       return topic;
    }
 }
```

```
// Демонстрация работы справочной системы на основе файла
class FileHelp {
   public static void main(String args[]) {
      Help hlpobj = new Help("helpfile.txt");
      String topic;

      System.out.println("Воспользуйтесь справочной системой.\n" +
                         "Для выхода из системы введите 'stop'.");

      do {
         topic = hlpobj.getSelection();

         if(!hlpobj.helpon(topic))
            System.out.println("Тема не найдена.\n");
      } while(topic.compareTo("stop") != 0);
   }
}
```

Спросим у эксперта

ВОПРОС. Имеется ли, помимо методов синтаксического анализа, определяемых в оболочках простых типов, другой простой способ преобразования числовой строки, вводимой с клавиатуры, в эквивалентную ей двоичную форму?

ОТВЕТ. Да, имеется. Другой способ преобразования числовой строки в ее внутреннее представление в двоичной форме состоит в использовании одного из методов, определенных в классе `Scanner` из пакета `java.util`. Этот класс читает данные, вводимые в удобном для чтения виде, преобразуя их в двоичную форму. Средствами класса `Scanner` можно организовать чтение данных, вводимых из самых разных источников, в том числе с консоли и из файлов. Следовательно, его можно использовать для чтения числовой строки, введенной с клавиатуры, присваивая полученное значение переменной. И хотя в классе `Scanner` содержится слишком много средств, чтобы описать их подробно, ниже приведены основные примеры его применения.

Для организации ввода с клавиатуры средствами класса `Scanner` необходимо сначала создать объект этого класса, связанный с потоком ввода с консоли. Для этой цели служит следующий конструктор:

```
Scanner(InputStream from)
```

Этот конструктор создает объект типа `Scanner`, который использует поток ввода, определяемый параметром *from*, в качестве источника ввода данных. С помощью этого конструктора можно создать объект типа `Scanner`, связанный с потоком ввода с консоли, как показано ниже:

```
Scanner conin = new Scanner(System.in);
```

Это оказывается возможным благодаря тому, что поток `System.in` является объектом типа `InputStream`. После выполнения этой строки кода перемен-

ную `conin` ссылки на объект типа `Scanner` можно использовать для чтения данных, вводимых с клавиатуры.

Как только будет создан объект типа `Scanner`, им нетрудно воспользоваться для чтения числовой строки, вводимой с клавиатуры. Ниже приведен общий порядок выполняемых для этого действий.

1. Определить, имеются ли вводимые данные конкретного типа, вызвав один из методов `hasNext`*X* класса `Scanner`, где *X* — нужный тип вводимых данных.
2. Если вводимые данные имеются, прочитать их, вызвав один из методов `next`*X* класса `Scanner`.

Как следует из приведенного выше порядка действий, в классе `Scanner` определены две группы методов, предназначенных для чтения вводимых данных. К первой из них относятся методы `hasNext`*X*, в том числе `hasNextInt()` и `hasNextDouble()`. Каждый из методов `hasNext`*X* возвращает логическое значение `true`, если очередной элемент данных, имеющийся в потоке ввода, относится к нужному типу данных, а иначе — логическое значение `false`. Так, логическое значение `true` возвращается при вызове метода `hasNextInt()` лишь в том случае, если очередной элемент данных в потоке ввода является целочисленным значением, представленным в удобном для чтения виде. Если данные нужного типа имеются в потоке ввода, их можно прочитать, вызвав один из методов класса `Scanner`, относящихся к группе `next`, например метод `nextInt()` или `nextDouble()`. Эти методы преобразуют данные соответствующего типа из удобной для чтения формы во внутреннее их представление в двоичном виде, возвращая полученный результат. Так, для чтения целочисленного значения, введенного с клавиатуры, следует вызвать метод `nextInt()`.

В приведенном ниже фрагменте кода показано, каким образом организуется чтение целочисленного значения с клавиатуры.

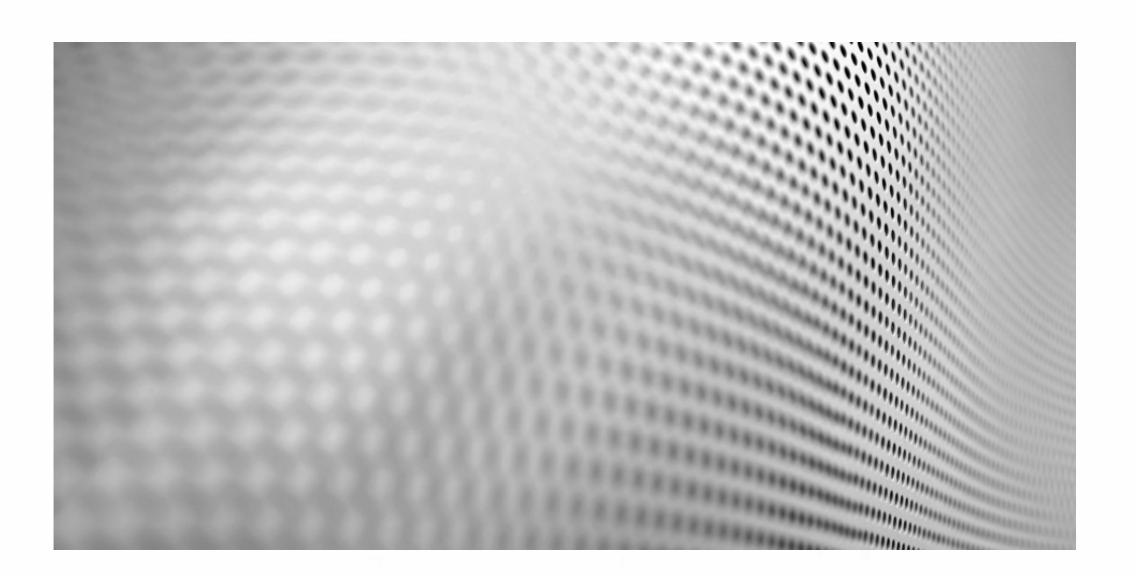
```
Scanner conin = new Scanner(System.in);
int i;

if (conin.hasNextInt()) i = conin.nextInt();
```

Если ввести с клавиатуры **123**, то в результате выполнения приведенного выше фрагмента кода переменная `i` будет содержать целочисленное значение `123`.

Формально методы из группы `next` можно вызывать без предварительного вызова методов из группы `hasNext`, но делать этого все же не рекомендуется. Ведь если методу из группы `next` не удастся обнаружить данные искомого типа, то он сгенерирует исключение `InputMismatchException`. Поэтому лучше сначала убедиться, что данные нужного типа имеются в потоке ввода, вызвав подходящий метод `hasNext`, и только после этого вызывать соответствующий метод `next`.

Вопросы и упражнения для самопроверки

1. Для чего в Java определены как байтовые, так и символьные потоки?
2. Как известно, консольные операции ввода-вывода осуществляются в текстовом виде. Почему же в Java для этой цели используются байтовые потоки?
3. Как открыть файл для чтения байтов?
4. Как открыть файл для чтения символов?
5. Как открыть файл для выполнения операций ввода-вывода с произвольным доступом?
6. Как преобразовать числовую строку `"123.23"` в ее двоичный эквивалент?
7. Напишите программу для копирования текстовых файлов. Видоизмените ее таким образом, чтобы все пробелы заменялись дефисами. Используйте при написании программы классы, представляющие байтовые потоки, а также традиционный способ закрытия файла явным вызовом метода `close()`.
8. Перепишите программу, созданную в предыдущем пункте, таким образом, чтобы в ней использовались классы, представляющие символьные потоки. На этот раз воспользуйтесь инструкцией `try` с ресурсами для автоматического закрытия файла.
9. К какому типу относится поток `System.in`?
10. Какое значение возвращает метод `read()` класса `InputStream` по достижении конца потока?
11. Поток какого типа используется для чтения двоичных данных?
12. Классы `Reader` и `Writer` находятся на вершине иерархии классов ____________________________.
13. Инструкция `try` с ресурсами служит для ________________________________.
14. Верно ли следующее утверждение: "Если для закрытия файла применяется традиционный способ, то лучше всего делать это в блоке `finally`"?



Глава 11

Многопоточное программирование

В этой главе...

- Общие сведения о многопоточной обработке
- Класс `Thread` и интерфейс `Runnable`
- Создание потока
- Создание нескольких потоков
- Определение момента завершения потока
- Приоритеты потоков
- Синхронизация потоков
- Использование синхронизированных методов
- Использование синхронизированных блоков кода
- Взаимодействие потоков
- Приостановка, возобновление и остановка потоков

Одной из наиболее впечатляющих новых возможностей Java можно по праву считать встроенную поддержку *многопоточного программирования*. Многопоточная программа состоит из двух или более частей, выполняемых параллельно. Каждая часть такой программы называется *потоком* и определяет отдельный путь выполнения команд. Таким образом, многопоточная обработка является одной из форм многозадачности.

Основы многопоточной обработки

Различают две разновидности многозадачности: на основе процессов и на основе потоков. В связи с этим важно понимать различия между ними. По сути, процесс представляет собой исполняемую программу. Поэтому *многозадачность на основе процессов* — это средство, обеспечивающее возможность выполнения на компьютере одновременно нескольких программ. Например, именно этот тип многозадачности позволяет вам запускать компилятор Java и в то же время работать с текстовым процессором, электронной таблицей или просматривать содержимое в Интернете. При организации многозадачности на основе процессов программа является наименьшей единицей кода, выполнение которой может координировать планировщик задач.

При организации *многозадачности на основе потоков* наименьшей единицей диспетчеризуемого кода является поток. Это означает, что в рамках одной программы могут выполняться одновременно несколько задач. Например, текстовый процессор может форматировать текст одновременно с его

выводом на печать, при условии, что оба этих действия выполняются в двух отдельных потоках. Несмотря на то что программы на Java выполняются в среде, поддерживающей многозадачность на основе процессов, в самих программах управлять процессами нельзя. Доступной остается только многозадачность на основе потоков.

Главное преимущество многопоточной обработки заключается в том, что она позволяет писать программы, которые работают очень эффективно благодаря использованию холостых периодов процессора, неизбежно возникающих в ходе выполнения большинства программ. Как известно, большинство устройств ввода-вывода, будь то устройства, подключенные к сетевым портам, дисковые накопители или клавиатура, работает намного медленнее, чем центральный процессор (ЦП). Поэтому бо́льшую часть своего времени программе приходится ждать отправки данных на устройство ввода-вывода или получения информации от него. Благодаря многопоточной обработке программа может решать какую-нибудь другую задачу во время вынужденного простоя процессора. Например, в то время как одна часть программы отправляет файл через соединение с Интернетом, другая ее часть может считывать текстовую информацию, вводимую с клавиатуры, а третья — осуществлять буферизацию очередного блока отправляемых данных.

Как вам, наверное, известно, за последние несколько лет широкое распространение получили многопроцессорные или многоядерные вычислительные системы, хотя по-прежнему повсеместно используются и однопроцессорные системы. В этой связи следует иметь в виду, что языковые средства организации многопоточной обработки в Java пригодны для обеих разновидностей вычислительных систем. В одноядерной системе параллельно выполняющиеся потоки разделяют ресурсы одного ЦП, получая по очереди квант его времени. Поэтому в одноядерной системе два или более потока на самом деле не выполняются одновременно, а лишь используют время простоя ЦП. С другой стороны, в многопроцессорных или многоядерных системах несколько потоков могут выполняться действительно одновременно. Это, как правило, позволяет повысить производительность программ и скорость выполнения отдельных операций.

Поток может находиться в одном из нескольких состояний. В целом поток может быть *выполняющимся*; *готовым к выполнению*, как только он получит время и ресурсы ЦП; *приостановленным*, т.е. временно не выполняющимся; *возобновленным* в дальнейшем; *заблокированным* в ожидании ресурсов для своего выполнения; а также *завершенным*, когда его выполнение закончено и не может быть возобновлено.

В связи с организацией многозадачности на основе потоков возникает потребность в особом режиме, который называется *синхронизацией* и позволяет координировать выполнение потоков строго определенным образом. Для такой синхронизации в Java предусмотрена отдельная подсистема, основные средства которой рассматриваются в этой главе.

Если вы пишете программы для операционных систем типа Windows, то принципы многопоточного программирования должны быть уже вам знакомы. Но то обстоятельство, что в Java возможности управления потоками включены в сам язык, упрощает организацию многопоточной обработки, поскольку избавляет от необходимости реализовывать ее во всех деталях.

Класс Thread и интерфейс Runnable

В основу системы многопоточной обработки в Java положены класс `Thread` и интерфейс `Runnable`, входящие в пакет `java.lang`. Класс `Thread` инкапсулирует поток исполнения. Для того чтобы образовать новый поток, нужно создать класс, являющийся подклассом `Thread` или реализующий интерфейс `Runnable`.

В классе `Thread` определен ряд методов, позволяющих управлять потоками. Некоторые из наиболее употребительных методов описаны ниже. По мере их представления в последующих примерах программ вы ознакомитесь с ними поближе.

Метод	Описание
`final String getName()`	Получает имя потока
`final int getPriority()`	Получает приоритет потока
`final boolean isAlive()`	Определяет, выполняется ли поток
`final void join()`	Ожидает завершения потока
`void run()`	Определяет точку входа в поток
`static void sleep(long` *`миллисекунды`*`)`	Приостанавливает выполнение потока на указанное число миллисекунд
`void start()`	Запускает поток, вызывая его метод `run()`

В каждом процессе имеется как минимум один поток выполнения, который называется ***основным потоком***. Он получает управление уже при запуске программы. Следовательно, во всех рассмотренных до сих пор примерах использовался основной поток. От основного потока могут быть порождены другие, подчиненные потоки.

Создание потока

Для того чтобы создать поток, нужно построить объект типа `Thread`. Класс `Thread` инкапсулирует объект, который может стать исполняемым. Как уже отмечалось, в Java пригодные для выполнения объекты можно создавать двумя способами:

- с помощью реализации интерфейса `Runnable`;
- путем создания подкласса класса `Thread`.

В большинстве примеров, представленных в этой главе, будет применяться первый способ. Тем не менее в упражнении 11.1 будет продемонстрировано, каким образом можно реализовать поток путем расширения класса `Thread`. В любом случае создание экземпляра потока, организация доступа к нему и управление потоком осуществляются средствами класса `Thread`. Единственное отличие обоих способов состоит в том, как создается класс, активизирующий поток.

Интерфейс `Runnable` дает абстрактное описание единицы исполняемого кода. Для формирования потока подходит любой объект, реализующий этот интерфейс. В интерфейсе `Runnable` объявлен только один метод — `run()`:

```
public void run()
```

В теле метода `run()` определяется код, соответствующий новому потоку. Из этого метода можно вызывать другие методы, использовать в нем различные классы и объявлять переменные точно так же, как это делается в основном потоке. Единственное отличие состоит в том, что метод `run()` создает точку входа в поток, выполняемый в программе параллельно с основным. Этот поток выполняется до тех пор, пока не произойдет возврат из метода `run()`.

После создания класса, реализующего интерфейс `Runnable`, следует создать экземпляр объекта типа `Thread` на основе объекта данного класса. В классе `Thread` определен ряд конструкторов. В дальнейшем будет использоваться следующий конструктор:

```
Thread(Runnable threadOb)
```

В качестве параметра *`threadOb`* этому конструктору передается экземпляр класса, реализующего интерфейс `Runnable`. Это позволяет определить, откуда начнется выполнение потока.

Созданный поток не запустится на выполнение до тех пор, пока не будет вызван метод `start()`, объявленный в классе `Thread`. В сущности, единственным назначением метода `start()` является вызов метода `run()`. Метод `start()` объявляется следующим образом:

```
void start()
```

Ниже приведен пример программы, в которой создается и запускается на выполнение новый поток.

```
// Создание потока путем реализации интерфейса Runnable

class MyThread implements Runnable {  // ← Объекты класса MyThread могут выполняться в своих собственных потоках, поскольку класс MyThread реализует интерфейс Runnable
   String thrdName;

   MyThread(String name) {
      thrdName = name;
   }

   // Точка входа в поток
   public void run() {  // ← Запуск потоков на выполнение
      System.out.println(thrdName + " - запуск");
```

```
      try {
         for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println("В " + thrdName + ", счетчик: " +
                               count);
         }
      }
      catch(InterruptedException exc) {
         System.out.println(thrdName + " - прерван");
      }
      System.out.println(thrdName + " - завершение");
   }
}

class UseThreads {
   public static void main(String args[]) {
      System.out.println("Запуск основного потока");

      // Сначала создать объект типа MyThread
      MyThread mt = new MyThread("Порожденный поток #1"); ◄── Создание выполняемого объе та

      // Затем сформировать поток на основе этого объекта
      Thread newThrd = new Thread(mt); ◄── Конструирование потока на основе этого объе та

      // Наконец, начать выполнение потока
      newThrd.start(); ◄── Запуск потока на выполнение

      for(int i=0; i<50; i++) {
         System.out.print(".");
         try {
            Thread.sleep(100);
         }
         catch(InterruptedException exc) {
            System.out.println("Прерывание основного потока");
         }
      }

      System.out.println("Завершение основного потока");
   }
}
```

Рассмотрим эту программу более подробно. Как видите, класс `MyThread` реализует интерфейс `Runnable`. Это означает, что объект типа `MyThread` подходит для использования в качестве потока, следовательно, его можно передать конструктору класса `Thread`.

В теле метода `run()` имеется цикл, счетчик которого принимает значения от 0 до 9. Обратите внимание на вызов метода `sleep()`. Этот метод приостанавливает поток, из которого он был вызван, на указанное число миллисекунд. Ниже приведена общая форма объявления данного метода.

```
static void sleep(long миллисекунды) throws InterruptedException
```

Единственный параметр метода `sleep()` задает время задержки, определяемое количеством миллисекунд. Как следует из объявления этого метода, в нем может быть сгенерировано исключение `InterruptedException`. Следовательно, его нужно вызывать в блоке `try`. Имеется и другой вариант метода `sleep()`, позволяющий точнее указывать время задержки в миллисекундах и дополнительно в наносекундах. Когда метод `sleep()` вызывается в методе `run()`, выполнение потока приостанавливается на 400 миллисекунд на каждом шаге цикла. Благодаря этому поток выполняется достаточно медленно, чтобы за ним можно было проследить.

В методе `main()` создается новый объект типа `Thread`. Для этой цели служит приведенная ниже последовательность инструкций.

```
// Сначала создать объект типа MyThread
MyThread mt = new MyThread("Порожденный поток #1");

// Затем сформировать поток на основе этого объекта
Thread newThrd = new Thread(mt);

// И наконец, начать выполнение потока
newThrd.start();
```

Как следует из комментариев к программе, сначала создается объект типа `MyThread`, который затем используется для создания объекта типа `Thread`. Его можно передать конструктору класса `Thread` в качестве параметра, поскольку класс `MyThread` реализует интерфейс `Runnable`. Наконец, для запуска нового потока вызывается метод `start()`, что приводит к вызову метода `run()` из порожденного потока. После вызова метода `start()` управление возвращается методу `main()`, где начинается выполнение цикла `for`. Этот цикл повторяется 50 раз, приостанавливая на 100 миллисекунд выполнение потока на каждом своем шаге. Оба потока продолжают выполняться, разделяя ресурсы ЦП в однопроцессорной системе до тех пор, пока циклы в них не завершатся. Ниже приведен результат выполнения данной программы. Вследствие отличий в вычислительных средах у вас может получиться несколько иной результат.

```
Запуск основного потока
Порожденный поток #1 - запуск
...В Порожденный поток #1, счетчик: 0
....В Порожденный поток #1, счетчик: 1
....В Порожденный поток #1, счетчик: 2
...В Порожденный поток #1, счетчик: 3
....В Порожденный поток #1, счетчик: 4
....В Порожденный поток #1, счетчик: 5
....В Порожденный поток #1, счетчик: 6
...В Порожденный поток #1, счетчик: 7
....В Порожденный поток #1, счетчик: 8
....В Порожденный поток #1, счетчик: 9
Порожденный поток #1 - завершение
................Завершение основного потока
```

В рассматриваемом здесь первом примере организации многопоточной обработки интерес представляет следующее обстоятельство: для демонстрации того факта, что основной и порожденный потоки выполняются одновременно, необходимо задержать завершение метода `main()` до тех пор, пока не закончится выполнение порожденного потока `mt`. В данном примере это достигается за счет использования отличий во временн*ы*х характеристиках обоих потоков. Вызовы метода `sleep()` из цикла `for` в методе `main()` приводят к суммарной задержке в 5 секунд (50 шагов цикла × 100 миллисекунд), тогда как суммарная задержка с помощью того же самого метода в аналогичном цикле в методе `run()` составляет лишь 4 секунды (10 шагов цикла × 400 миллисекунд). Поэтому метод `run()` завершится приблизительно на секунду раньше, чем метод `main()`. В итоге основной и порожденный потоки будут выполняться параллельно до тех пор, пока не завершится дочерний поток `mt`. После этого, приблизительно через одну секунду, завершится и основной поток в методе `main()`.

Различий во временн*ы*х характеристиках обоих потоков в данном и ряде последующих простых примеров хватает для того, чтобы основной поток в методе `main()` завершился последним, но на практике этого, как правило, оказывается недостаточно. В Java предоставляются более совершенные способы, позволяющие организовать ожидание завершения потока. Далее будет продемонстрирован более совершенный способ организации ожидания одним потоком завершения другого.

И последнее замечание: обычно многопоточная программа разрабатывается с таким расчетом, чтобы последним завершал свою работу основной поток. Как правило, выполнение программы продолжается до тех пор, пока все потоки не завершат работу. Поэтому завершение основного потока последним является не требованием, а рекомендуемой практикой, особенно для тех, кто лишь начинает осваивать многопоточное программирование.

Спросим у эксперта

ВОПРОС. Почему основной поток рекомендуется завершать последним?

ОТВЕТ. В основном потоке удобно выполнять действия по подготовке к завершению программы, например закрывать файлы. Именно поэтому основной поток желательно завершать последним. К счастью, организовать ожидание в основном потоке завершения порожденных потоков совсем не сложно.

Несложные усовершенствования многопоточной программы

В предыдущей программе были продемонстрированы основы создания класса `Thread` на основе интерфейса `Runnable` с последующим запуском потока. Подход, использованный при создании этой программы, абсолютно верен и дает нужные результаты. Но если внести два небольших изменения, то в

некоторых случаях класс MyThread станет более гибким и проще в применении. Более того, эти изменения будут весьма полезными при создании собственных классов Runnable. Если же вы внесете одно существенное изменение в класс MyThread, то получите еще один бонус в классе Thread. Начнем с рассмотрения этого изменения.

Заметьте, что в предыдущей программе переменная экземпляра thrdName определена в классе MyThread и используется для хранения имени потока. Но на самом деле нет никакой необходимости хранить название потока в классе MyThread, поскольку его можно присвоить потоку в случае его создания. Для этого используется следующая версия конструктора Thread:

```
Thread(Runnable threadOb, String имя)
```

Здесь параметр *имя* содержит имя потока. Чтобы получить имя потока, можно вызвать метод getName(), определенный в классе Thread. Общая форма этого метода приведена ниже.

```
final String getName()
```

Благодаря присваиванию имени потоку во время его создания обеспечиваются два преимущества. Во-первых, не нужно использовать отдельную переменную для хранения имени, поскольку с этой задачей справляется класс Thread. Во-вторых, имя потока будет доступно любому коду, который хранит ссылку на этот поток. Также можно присвоить имя потоку после его создания, используя метод setName():

```
final void setName(String threadName)
```

где параметр *threadName* задает новое имя потока.

Как уже упоминалось, два изменения могут, в зависимости от ситуации, сделать класс MyThread более удобным в применении. Во-первых, конструктор MyThread может создать объект Thread для потока и сохранить ссылку на этот поток в переменной экземпляра. Используя этот подход, поток будет готов к старту сразу же после завершения конструктора MyThread. Можно просто вызвать метод start() для экземпляра класса Thread, инкапсулированного в класс MyThread.

Благодаря второму изменению поток может начать выполняться сразу же после создания. Это полезно в тех случаях, когда не нужно отделять этап создания потока от этапа его выполнения. Чтобы воспользоваться этим подходом для класса MyThread, можно применить статический ***фабричный метод***, который:

1) создает новый экземпляр класса MyThread;
2) вызывает метод потока start(), связанный с этим экземпляром класса;
3) возвращает ссылку на только что созданный объект MyThread.

Такой подход позволяет создавать и запускать поток с помощью единственного вызова метода. Благодаря этому упрощается класс MyThread, особенно в тех случаях, когда нужно создать и запустить несколько потоков.

В этой версии предыдущей программы были учтены только что внесенные изменения.

```
// Изменения класса MyThread. Эта версия класса MyThread
// создает объект Thread путем вызова его конструктора
// и сохранения в переменной экземпляра thrd.
// Также присваивается имя потоку и используется
// фабричный метод для создания и запуска потока.

class MyThread implements Runnable {
   Thread thrd;  ◄—— Ссылка на поток хранится в переменной thrd

   // Создание нового потока на основе интерфейса и
   // присваивание ему имени
   MyThread(String name) {
      thrd = new Thread(this, name);  ◄—— Имя потоку присваивается при его создании
   }

   // Создание и запуск потока с помощью фабричного метода
   public static MyThread createAndStart(String name) {
      MyThread myThrd = new MyThread(name);

      myThrd.thrd.start(); // запуск потока  ◄—— Начало выполнения потока
      return myThrd;
   }

   // Точка входа для потока
   public void run() {
      System.out.println(thrd.getName() + " - запуск.");
      try {
         for(int count=0; count<10; count++) {
            Thread.sleep(400);
            System.out.println("В " + thrd.getName() +
                               ", счетчик: " + count);
         }
      }
      catch(InterruptedException exc) {
         System.out.println(thrd.getName() + " - прерван.");
      }
      System.out.println(thrd.getName() + " - завершение.");
   }
}

class ThreadVariations {
   public static void main(String args[]) {
      System.out.println("Запуск основного потока.");

      // Создание и запуск потока
      MyThread mt = MyThread.createAndStart("Порожденный поток #1");
                                      ▲
                                      └—— Запуск нового потока после его создания
```

```
        for(int i=0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            }
            catch(InterruptedException exc) {
                System.out.println("Прерывание основного потока.");
            }
        }
        System.out.println("Завершение основного потока.");
    }
}
```

Эта версия программы дает тот же результат, что и предыдущая, но на этот раз класс `MyThread` не содержит имя потока. Вместо этого в переменной экземпляра `thrd` хранится ссылка на объект `Thread`, созданный конструктором `MyThread`.

```
MyThread(String name) {
    thrd = new Thread(this, name);
}
```

После вызова конструктора `MyThread` переменная `thrd` будет содержать ссылку на только что созданный поток. Для запуска потока на выполнение достаточно вызвать метод `start()` вместе с переменной `thrd`.

А сейчас обратим внимание на фабричный метод `createAndStart()`, код которого приведен ниже.

```
// Создание и запуск потока с помощью фабричного метода
public static MyThread createAndStart(String name) {
    MyThread myThrd = new MyThread(name);

    myThrd.thrd.start(); // запуск потока
    return myThrd;
}
```

После вызова этого метода создается новый экземпляр класса `MyThread` под названием `myThrd`, а затем вызывается метод `start()` для копии `myThrd` переменной `thrd`. И напоследок возвращается ссылка на только что созданный экземпляр `MyThread`. Таким образом, как только возвращается вызов метода `createAndStart()`, поток уже будет запущен. В результате эта строка создает и начинает выполнение потока за один вызов в методе `main()`:

```
MyThread mt = MyThread.createAndStart("Порожденный поток#1");
```

В силу удобств, предлагаемых методом `createAndStart()`, он будет применяться в нескольких примерах этой главы. Более того, его можно адаптировать для использования в ваших собственных многопоточных приложениях. Конечно, в тех случаях, когда выполнение потока отделено от его создания, можно просто создать объект `MyThread`, а позднее вызвать метод `start()`.

Спросим у эксперта

ВОПРОС. Ранее был использован термин *фабричный метод* и продемонстрирован один пример этого метода под названием `createAndStart()`. Существует ли более общее определение такого метода?

ОТВЕТ. Да. Согласно общему определению, фабричный метод — это метод, который возвращает объект класса. Обычно фабричные методы являются статическими методами класса и могут использоваться во многих ситуациях. Вот несколько примеров. Как вы только что видели в случае с методом `createAndStart()`, фабричный метод позволяет создать объект и назначить ему некоторое специфическое состояние до возврата этого объекта вызывающему методу. Еще один тип фабричного метода используется для создания легкого для запоминания имени, которое указывает на разновидность создаваемого объекта.

Например, для класса `Line` могут применяться фабричные методы, такие как `createRedLine()` или `createBlueLine()`, которые создают линии определенных цветов. Вместо запоминания потенциально сложного вызова конструктора можно просто воспользоваться фабричным методом, имя которого указывает на тип требуемой линии. В некоторых случаях фабричный метод может не создавать новый объект, а повторно использовать прежний. По мере изучения Java вы узнаете о том, что фабричные объекты часто применяются в Java API.

Упражнение 11.1

Расширение класса Thread

`ExtendThread.java`

Реализация интерфейса `Runnable` — это лишь один из способов получения экземпляров потоковых объектов. Другой способ состоит в создании подкласса, производного от класса `Thread`. В этом упражнении будет продемонстрировано, каким образом расширение класса `Thread` позволяет реализовать такие же функциональные возможности, как и рассмотренная в начале главы программа `UseThreads`.

Класс, расширяющий класс `Thread`, должен переопределить метод `run()`, который является точкой входа в новый поток. Для того чтобы начать выполнение нового потока, следует вызвать метод `start()`. Можно также переопределить и другие методы класса `Thread`, но делать это не обязательно. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл `ExtendThread.java`. Начните этот файл следующими строками кода.

```
/*
   Упражнение 11.1
```

```
   Расширение класса Thread.
*/
class MyThread extends Thread {
```

Обратите внимание на то, что класс MyThread расширяет класс Thread вместо реализации интерфейса Runnable.

2. Добавьте следующий конструктор MyThread.

```
// Конструктор нового потока
MyThread(String name) {
   super(name); // имя потока
}
```

Ключевое слово super используется для вызова следующей версии конструктора Thread:

```
Thread(String threadName)
```

Параметр *threadName* определяет имя потока. Как уже объяснялось, класс Thread обеспечивает возможность хранения имени потока. Таким образом, переменная экземпляра не требуется для хранения имени потока в классе MyThread.

3. Завершите класс MyThread, добавив следующий метод run().

```
   // Точка входа для потока
   public void run() {
      System.out.println(getName() + " - запуск.");
      try {
         for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println("В " + getName() + ", счетчик: " +
                               count);
         }
      }
      catch(InterruptedException exc) {
         System.out.println(getName() + " - прерван.");
      }

      System.out.println(getName() + " - завершение.");
   }
}
```

Обратите внимание на вызовы метода getName(). Поскольку класс Extend Thread расширяет класс Thread, можно непосредственно вызвать все методы класса Thread, включая метод getName().

4. Добавьте класс ExtendThread, показанный ниже.

```
class ExtendThread {
   public static void main(String args[]) {
      System.out.println("Запуск основного потока.");

      MyThread mt = new MyThread("Порожденный поток #1");

      mt.start();
```

```
      for(int i=0; i < 50; i++) {
         System.out.print(".");
         try {
            Thread.sleep(100);
         }
         catch(InterruptedException exc) {
            System.out.println("Прерывание основного потока.");
         }
      }

      System.out.println("Завершение основного потока.");
   }
}
```

Заметьте, как в методе `main()` создается экземпляр класса `MyThread`, который затем запускается с помощью следующих двух строк кода.

```
MyThread mt = new MyThread("Порожденный поток #1");
mt.start();
```

Поскольку класс `MyThread` реализуется объект `Thread`, метод `start()` вызывается непосредственно из экземпляра класса `MyThread`, `mt`.

5. Ниже приведен завершенный код программы. Результат выполнения этой программы такой же, как и результат выполнения программы `UseThreads`, но в данном случае расширяется класс `Thread` вместо реализации интерфейса `Runnable`.

```
/*
   Упражнение 11.1

   Расширение класса Thread.
*/
class MyThread extends Thread {

   // Конструктор нового потока
   MyThread(String name) {
      super(name); // имя потока
   }

   // Точка входа для потока
   public void run() {
      System.out.println(getName() + " - запуск.");
      try {
         for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println("В " + getName() + ", счетчик: " +
                               count);
         }
      }
      catch(InterruptedException exc) {
         System.out.println(getName() + " - прерван.");
      }
```

```
      System.out.println(getName() + " - завершение.");
   }
}

class ExtendThread {
   public static void main(String args[]) {
      System.out.println("Запуск основного потока.");

      MyThread mt = new MyThread("Порожденный поток #1");

      mt.start();

      for(int i=0; i < 50; i++) {
         System.out.print(".");
         try {
            Thread.sleep(100);
         }
         catch(InterruptedException exc) {
            System.out.println("Прерывание основного потока.");
         }
      }

      System.out.println("Завершение основного потока.");
   }
}
```

6. В процессе расширения класса Thread можно также включить возможность создания и запуска потока за один шаг с помощью статичного фабричного метода, подобного методу, используемому в ранее показанной программе ThreadVariations. Чтобы воспользоваться этой возможностью, добавьте в класс MyThread следующий метод.

```
public static MyThread createAndStart(String name) {
   MyThread myThrd = new MyThread(name);

   myThrd.start();
   return myThrd;
}
```

Как видите, этот метод создает новый экземпляр класса MyThread с указанным именем, вызывает метод start() в данном потоке, а затем возвращает ссылку на поток. Чтобы создать метод createAndStart(), замените следующие две строки кода в методе main():

```
System.out.println("Запуск основного потока.");
MyThread mt = new MyThread("Порожденный поток #1");
```

строкой

```
MyThread mt = MyThread.createAndStart("Порожденный поток #1");
```

После внесения этих изменений программа будет выполняться как и раньше, но создание и запуск потока будут происходить с помощью единственного вызова метода.

Создание нескольких потоков

В предыдущем примере был создан только один порожденный поток. Но в программе можно породить столько потоков, сколько требуется. Например, в приведенной ниже программе формируются три порожденных потока.

```
// Создание нескольких потоков

class MyThread implements Runnable {
   Thread thrd;

   // Конструктор нового потока
   MyThread(String name) {
      thrd = new Thread(this, name);
   }

   // Создание и запуск потока с помощью фабричного метода
   public static MyThread createAndStart(String name) {
      MyThread myThrd = new MyThread(name);

      myThrd.thrd.start(); // запуск потока
      return myThrd;
   }

   // Точка входа для потока
   public void run() {
      System.out.println(thrd.getName() + " - запуск.");
      try {
         for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println("В " + thrd.getName() +
                                 ", счетчик: " + count);
         }
      }
      catch(InterruptedException exc) {
         System.out.println(thrd.getName() + " - прерван.");
      }
      System.out.println(thrd.getName() + " - завершение.");
   }
}

class MoreThreads {
   public static void main(String args[]) {
      System.out.println("Запуск основного потока.");

      MyThread mt1 = MyThread.createAndStart("Порожденный поток #1");
      MyThread mt2 = MyThread.createAndStart("Порожденный поток #2"); ◄── Создание и запуск трех потоков
      MyThread mt3 = MyThread.createAndStart("Порожденный поток #3");

      for(int i=0; i < 50; i++) {
         System.out.print(".");
```

```
            try {
               Thread.sleep(100);
            }
            catch(InterruptedException exc) {
               System.out.println("Прерывание основного потока.");
            }
         }

         System.out.println("Завершение основного потока.");
   }
}
```

Ниже приведен результат выполнения данной программы.

```
Запуск основного потока
Порожденный поток #1 - запуск
Порожденный поток #2 - запуск
Порожденный поток #3 - запуск
...В Порожденный поток #3, счетчик: 0
В Порожденный поток #2, счетчик: 0
В Порожденный поток #1, счетчик: 0
....В Порожденный поток #1, счетчик: 1
В Порожденный поток #2, счетчик: 1
В Порожденный поток #3, счетчик: 1
....В Порожденный поток #2, счетчик: 2
В Порожденный поток #3, счетчик: 2
В Порожденный поток #1, счетчик: 2
...В Порожденный поток #1, счетчик: 3
В Порожденный поток #2, счетчик: 3
В Порожденный поток #3, счетчик: 3
....В Порожденный поток #1, счетчик: 4
В Порожденный поток #3, счетчик: 4
В Порожденный поток #2, счетчик: 4
....В Порожденный поток #1, счетчик: 5
В Порожденный поток #3, счетчик: 5
В Порожденный поток #2, счетчик: 5
...В Порожденный поток #3, счетчик: 6
.В Порожденный поток #2, счетчик: 6
В Порожденный поток #1, счетчик: 6
...В Порожденный поток #3, счетчик: 7
В Порожденный поток #1, счетчик: 7
В Порожденный поток #2, счетчик: 7
....В Порожденный поток #2, счетчик: 8
В Порожденный поток #1, счетчик: 8
В Порожденный поток #3, счетчик: 8
....В Порожденный поток #1, счетчик: 9
Порожденный поток #1 - завершение
В Порожденный поток #2, счетчик: 9
Порожденный поток #2 - завершение
В Порожденный поток #3, счетчик: 9
Порожденный поток #3 - завершение
............Завершение основного потока
```

Как видите, после запуска все три потока совместно используют ресурсы ЦП. Следует иметь в виду, что потоки в данном примере запускаются на выполнение в том порядке, в каком они были созданы. Но так происходит не всегда. Исполняющая среда Java сама планирует выполнение потоков. Вследствие отличий в вычислительных средах у вас может получиться несколько иной результат.

Определяем момент завершения потока

Нередко требуется знать, когда завершится поток. Так, в приведенных выше примерах ради большей наглядности нужно было поддерживать основной поток действующим до тех пор, пока не завершатся остальные потоки. Для этой цели основной поток переводился в состояние ожидания на более продолжительное время, чем порожденные им потоки. Но такое решение вряд ли можно считать удовлетворительным или общеупотребительным.

Правда, в классе Thread предусмотрены два средства, позволяющие определить, завершился ли поток. Первым из них является метод isAlive(), объявление которого приведено ниже.

```
final boolean isAlive()
```

Этот метод возвращает значение true, если поток, для которого он вызывается, все еще выполняется. В противном случае он возвращает значение false. Для того чтобы опробовать метод isAlive() на практике, замените в предыдущей программе класс MoreThreads новой версией, исходный код которой приведен ниже.

```
// Использование метода isAlive().
class MoreThreads {
   public static void main(String args[]) {
      System.out.println("Запуск основного потока");

      MyThread mt1 = new MyThread("Порожденный поток #1");
      MyThread mt2 = new MyThread("Порожденный поток #2");
      MyThread mt3 = new MyThread("Порожденный поток #3");

      do {
         System.out.print(".");
         try {
            Thread.sleep(100);
         }
         catch(InterruptedException exc) {
            System.out.println("Прерывание основного потока");
         }
      } while (mt1.thrd.isAlive() ||
               mt2.thrd.isAlive() ||   ◄—— Ожидание завершения всех потоков
               mt3.thrd.isAlive());
      System.out.println("Завершение основного потока");
   }
}
```

Эта версия дает тот же результат, что и предыдущая. Единственное отличие состоит в том, что в данном случае ожидание завершения порожденного потока организовано с помощью метода isAlive(). Вторым средством, позволяющим определить, завершился ли поток, является метод join(), объявление которого приведено ниже.

```
final void join() throws InterruptedException
```

Этот метод ожидает завершения потока, для которого он был вызван. Выбор его имени был обусловлен тем, что вызывающий поток ожидает, когда указанный поток присоединится к нему. Имеется и другой вариант метода join(), позволяющий указать максимальное время ожидания момента завершения потока.

В приведенном ниже примере программы наличие метода join() гарантирует, что основной поток завершит работу последним.

```
// Использование метода join()
class MyThread implements Runnable {
   Thread thrd;

   // Конструктор нового потока
   MyThread(String name) {
      thrd = new Thread(this, name);
   }

   // Создание и запуск потока с помощью фабричного метода
   public static MyThread createAndStart(String name) {
      MyThread myThrd = new MyThread(name);

      myThrd.thrd.start(); // запуск потока
      return myThrd;
   }

   // Точка входа для потока
   public void run() {
      System.out.println(thrd.getName() + " - запуск.");
      try {
         for(int count=0; count < 10; count++) {
            Thread.sleep(400);
            System.out.println("В " + thrd.getName() +
                               ", счетчик: " + count);
         }
      }
      catch(InterruptedException exc) {
         System.out.println(thrd.getName() + " - прерван.");
      }

      System.out.println(thrd.getName() + " - завершение.");
   }
}
```

```
class JoinThreads {
   public static void main(String args[]) {
      System.out.println("Запуск основного потока.");

      MyThread mt1 = MyThread.createAndStart("Порожденный поток #1");
      MyThread mt2 = MyThread.createAndStart("Порожденный поток #2");
      MyThread mt3 = MyThread.createAndStart("Порожденный поток #3");

      try {
         mt1.thrd.join();   // Ожидание завершения указанного потока
         System.out.println("Порожденный поток #1 - присоединен.");
         mt2.thrd.join();
         System.out.println("Порожденный поток #2 - присоединен.");
         mt3.thrd.join();
         System.out.println("Порожденный поток #3 - присоединен.");
      }
      catch(InterruptedException exc) {
         System.out.println("Прерывание основного потока.");
      }

      System.out.println("Завершение основного потока.");
   }
}
```

Результат выполнения данной программы приведен ниже. Вследствие отличий в вычислительных средах он может получиться у вас несколько иным.

```
Запуск основного потока
Порожденный поток #1 - запуск
Порожденный поток #2 - запуск
Порожденный поток #3 - запуск
В Порожденный поток #2, счетчик: 0
В Порожденный поток #1, счетчик: 0
В Порожденный поток #3, счетчик: 0
В Порожденный поток #2, счетчик: 1
В Порожденный поток #3, счетчик: 1
В Порожденный поток #1, счетчик: 1
В Порожденный поток #2, счетчик: 2
В Порожденный поток #1, счетчик: 2
В Порожденный поток #3, счетчик: 2
В Порожденный поток #2, счетчик: 3
В Порожденный поток #3, счетчик: 3
В Порожденный поток #1, счетчик: 3
В Порожденный поток #3, счетчик: 4
В Порожденный поток #2, счетчик: 4
В Порожденный поток #1, счетчик: 4
В Порожденный поток #3, счетчик: 5
В Порожденный поток #1, счетчик: 5
В Порожденный поток #2, счетчик: 5
В Порожденный поток #3, счетчик: 6
В Порожденный поток #2, счетчик: 6
В Порожденный поток #1, счетчик: 6
```

```
В Порожденный поток #3, счетчик: 7
В Порожденный поток #1, счетчик: 7
В Порожденный поток #2, счетчик: 7
В Порожденный поток #3, счетчик: 8
В Порожденный поток #2, счетчик: 8
В Порожденный поток #1, счетчик: 8
В Порожденный поток #3, счетчик: 9
Порожденный поток #3 - завершение
В Порожденный поток #2, счетчик: 9
Порожденный поток #2 - завершение
В Порожденный поток #1, счетчик: 9
Порожденный поток #1 - завершение
Порожденный поток #1 - присоединен
Порожденный поток #2 - присоединен
Порожденный поток #3 - присоединен
Завершение основного потока
```

Как видите, после того как вызываемый метод `join()` возвращает управление, выполнение потока прекращается.

Приоритеты потоков

С каждым потоком ассоциируется определенный приоритет. В частности, от приоритета потока зависит относительная доля процессорного времени, предоставляемого данному потоку, по сравнению с остальными активными потоками. Вообще говоря, в течение определенного промежутка времени низкоприоритетные потоки будут получать меньше времени центрального процессора (ЦП), а высокоприоритетные потоки — больше. Как и можно было ожидать, время ЦП, получаемое потоком, оказывает определяющее влияние на характеристики его выполнения и взаимодействия с другими потоками, выполняющимися в настоящий момент в системе.

Следует иметь в виду, что, помимо приоритета, на частоту доступа потока к ЦП оказывают влияние и другие факторы. Так, если высокоприоритетный поток ожидает доступа к некоторому ресурсу, например для ввода с клавиатуры, то он блокируется, и вместо него выполняется низкоприоритетный поток. Но когда высокоприоритетный поток получит доступ к ресурсам, он прервет низкоприоритетный поток и возобновит свое выполнение. На планирование работы потоков также влияет то, каким именно образом в операционной системе поддерживается многозадачность (см. врезку “Спросим у эксперта” в конце раздела). Следовательно, если один поток имеет более высокий приоритет, чем другой, это еще не означает, что первый поток будет исполняться быстрее второго. Высокий приоритет потока лишь означает, что потенциально он может получить больше времени ЦП.

При запуске порожденного потока его приоритет устанавливается равным приоритету родительского потока. Изменить приоритет можно, вызвав метод `setPriority()` класса `Thread`. Ниже приведено объявление этого метода.

```
final void setPriority(int уровень)
```

С помощью параметра *уровень* данному методу передается новый приоритет потока. Значение параметра *уровень* должно находиться в пределах от MIN_PRIORITY до MAX_PRIORITY. В настоящее время этим константам соответствуют числовые значения от 1 до 10. Для того чтобы восстановить приоритет потока, заданный по умолчанию, следует указать значение 5, которому соответствует константа NORM_PRIORITY. Константы, определяющие приоритеты потоков, определены как static final в классе Thread.

Получить текущий приоритет можно с помощью метода getPriority() класса Thread, объявляемого следующим образом:

```
final int getPriority()
```

Ниже приведен пример программы, демонстрирующий использование потоков с разным приоритетом. Потоки создаются как экземпляры класса Priority. В методе run() содержится цикл, отсчитывающий число своих шагов. Этот цикл завершает работу, когда значение счетчика достигает 10000000 или же когда статическая переменная stop принимает значение true. Первоначально переменной stop присваивается значение false, но первый же поток, заканчивающий отсчет, присваивает этой переменной значение true. В результате второй поток завершится, как только ему будет выделен квант времени. В цикле выполняется проверка символьной строки в переменной currentName на предмет совпадения с именем выcполняемого потока. Если они не совпадают, значит, произошло переключение задач. При этом отображается имя нового потока, которое присваивается переменной currentName. Это дает возможность следить за тем, насколько часто каждый поток получает время ЦП. После остановки обоих потоков выводится число шагов, выполненных в каждом цикле.

```
// Демонстрация потоков с разными приоритетами
class Priority implements Runnable {
   int count;
   Thread thrd;

   static boolean stop = false;
   static String currentName;

   // Конструктор нового потока
   Priority(String name) {
      thrd = new Thread(this, name);
      count = 0;
      currentName = name;
   }

   // Точка входа для потока
   public void run() {
      System.out.println(thrd.getName() + " - запуск.");
      do {
         count++;

         if(currentName.compareTo(thrd.getName()) != 0) {
            currentName = thrd.getName();
```

```
            System.out.println("В " + currentName);
         }
      } while(stop == false && count < 10000000);
      stop = true;

      System.out.println("\n" + thrd.getName() +
                         " - прерывание.");
   }
}

class PriorityDemo {
   public static void main(String args[]) {
      Priority mt1 = new Priority("Высокий приоритет");
      Priority mt2 = new Priority("Низкий приоритет");
      Priority mt3 = new Priority("Обычный приоритет #1");
      Priority mt4 = new Priority("Обычный приоритет #2");
      Priority mt5 = new Priority("Обычный приоритет #3");

      // Присваивание приоритетов
      mt1.thrd.setPriority(Thread.NORM_PRIORITY+2);
      mt2.thrd.setPriority(Thread.NORM_PRIORITY-2);
      // Потоки mt3, mt4 и mt5 имеют обычный приоритет,
      // заданный по умолчанию

      // Запуск потоков
      mt1.thrd.start();
      mt2.thrd.start();
      mt3.thrd.start();
      mt4.thrd.start();
      mt5.thrd.start();

      try {
         mt1.thrd.join();
         mt2.thrd.join();
         mt3.thrd.join();
         mt4.thrd.join();
         mt5.thrd.join();
      }
      catch(InterruptedException exc) {
         System.out.println("Прерван основной поток.");
      }
      System.out.println("\nСчетчик потока с высоким приоритетом: " +
                         mt1.count);
      System.out.println("Счетчик потока с низким приоритетом: " +
                         mt2.count);
      System.out.println("Счетчик 1-го потока с обычным
                         приоритетом: " + mt3.count);
      System.out.println("Счетчик 2-го потока с обычным
                         приоритетом: " + mt4.count);
      System.out.println("Счетчик 3-го потока с обычным
                         приоритетом: " + mt5.count);
   }
}
```

Первый же поток, в котором достигнуто значение 10000000, завершает остальные потоки

Поток mt1 получает более высокий приоритет, чем поток mt2

Результат выполнения данной программы выглядит следующим образом:

```
Счетчик потока с высоким приоритетом: 10000000
Счетчик потока с высоким приоритетом: 3477862
Счетчик первого потока с обычным приоритетом: 7000045
Счетчик второго потока с обычным приоритетом: 6576054
Счетчик третьего потока с обычным приоритетом: 7373846
```

В данном примере большую часть времени ЦП получает высокоприоритетный поток. Очевидно, что результат выполнения программы существенно зависит от быстродействия ЦП, количества процессоров, типа операционной системы и наличия прочих задач, выполняющихся в системе.

Спросим у эксперта

ВОПРОС. Оказывает ли влияние конкретная реализация многозадачности на то, какую долю времени ЦП получает поток?

ОТВЕТ. Если не учитывать приоритет потока, то наиболее важным фактором, оказывающим влияние на выполнение потока, является способ реализации многозадачности и планирования заданий в операционной системе. В одних операционных системах применяется вытесняющая многозадачность, т.е. каждый поток получает квант времени. В других системах планирование задач осуществляется по-другому. В частности, второй поток иногда получает управление только после завершения первого. В системах с невытесняющей многозадачностью один поток будет господствовать над остальными, препятствуя их выполнению.

Синхронизация

При использовании нескольких потоков иногда возникает необходимость в координации их выполнения. Процесс, посредством которого это достигается, называют *синхронизацией*. Чаще всего синхронизацию используют в тех случаях, когда несколько потоков должны разделять ресурс, который может быть одновременно доступен только одному потоку. Например, когда в одном потоке выполняется запись информации в файл, второму потоку должно быть запрещено выполнять это действие в тот же самый момент времени. Синхронизация требуется и тогда, когда один поток ожидает событие, вызываемое другим потоком. В подобных ситуациях требуются средства, позволяющие приостановить один из потоков до тех пор, пока не произойдет определенное событие в другом потоке. После этого ожидающий поток может возобновить свое выполнение.

Главным для синхронизации в Java является понятие *монитора*, контролирующего доступ к объекту. Монитор реализует принцип *блокировки*. Если объект заблокирован одним потоком, то он оказывается недоступным для других потоков. В какой-то момент объект разблокируется, благодаря чему другие потоки смогут получить к нему доступ.

У каждого объекта в Java имеется свой монитор. Этот механизм встроен в сам язык. Следовательно, синхронизировать можно любой объект. Для поддержки синхронизации в Java предусмотрено ключевое слово `synchronized` и ряд специальных методов, имеющихся у каждого объекта. А поскольку средства синхронизации встроены в сам язык, то пользоваться ими на практике очень просто — гораздо проще, чем может показаться на первый взгляд. Для многих программ средства синхронизации объектов, по сути, прозрачны.

Синхронизировать код можно двумя способами. Оба способа рассматриваются ниже, и в обоих используется ключевое слово `synchronized`.

Использование синхронизированных методов

Для того чтобы синхронизировать метод, в его объявлении следует указать ключевое слово `synchronized`. Когда такой метод получает управление, вызывающий поток активизирует монитор, что приводит к блокированию объекта. Если объект блокирован, он недоступен из другого потока, а кроме того, его нельзя вызвать из других синхронизированных методов, определенных в классе данного объекта. Когда выполнение синхронизированного метода завершается, монитор разблокирует объект, что позволяет другому потоку использовать этот метод. Таким образом, для достижения синхронизации программисту не приходится прилагать каких-то особых усилий.

Ниже приведен пример программы, демонстрирующий контролируемый доступ к методу `sumArray()`. Этот метод суммирует элементы целочисленного массива.

```
// Использование ключевого слова synchronized для
// управления доступом

class SumArray {
   private int sum;

   synchronized int sumArray(int nums[]) {  ◄── Метод sumArray() синхронизирован
      sum = 0; // обнуление суммы

      for(int i=0; i<nums.length; i++) {
         sum += nums[i];
         System.out.println("Текущее значение суммы для " +
                            Thread.currentThread().getName() +
                            " будет " + sum);
         try {
            Thread.sleep(10); // разрешение переключения
                              // между задачами
         }
         catch(InterruptedException exc) {
            System.out.println("Поток прерван.");
         }
      }
```

```
      return sum;
   }
}

class MyThread implements Runnable {
   Thread thrd;
   static SumArray sa = new SumArray();
   int a[];
   int answer;

   // Конструктор нового потока
   MyThread(String name, int nums[]) {
      thrd = new Thread(this, name);
      a = nums;
   }

   // Создание и запуск потока с помощью фабричного метода
   public static MyThread createAndStart(String name,
                                         int nums[]) {
      MyThread myThrd = new MyThread(name, nums);

      myThrd.thrd.start(); // запуск потока
      return myThrd;
   }

   // Точка входа для потока
   public void run() {
      int sum;

      System.out.println(thrd.getName() + " - запуск.");

      answer = sa.sumArray(a);
      System.out.println("Сумма для " + thrd.getName() +
                         " будет " + answer);

      System.out.println(thrd.getName() + " - завершение.");
   }
}

class Sync {
   public static void main(String args[]) {
      int a[] = {1, 2, 3, 4, 5};

      MyThread mt1 = MyThread.createAndStart("Порожденный
                                              поток #1", a);
      MyThread mt2 = MyThread.createAndStart("Порожденный
                                              поток #2", a);
      try {
         mt1.thrd.join();
         mt2.thrd.join();
      }
```

```
      catch(InterruptedException exc) {
         System.out.println("Прерывание основного потока.");
      }
   }
}
```

Выполнение этой программы дает следующий результат.

```
Порожденный поток #1 - запуск
Текущее значение суммы для Порожденный поток #1: 1
Порожденный поток #2 - запуск
Текущее значение суммы для Порожденный поток #1: 3
Текущее значение суммы для Порожденный поток #1: 6
Текущее значение суммы для Порожденный поток #1: 10
Текущее значение суммы для Порожденный поток #1: 15
Сумма для Порожденный поток #1: 15
Порожденный поток #1 - завершение
Текущее значение суммы для Порожденный поток #2: 1
Текущее значение суммы для Порожденный поток #2: 3
Текущее значение суммы для Порожденный поток #2: 6
Текущее значение суммы для Порожденный поток #2: 10
Текущее значение суммы для Порожденный поток #2: 15
Сумма для Порожденный поток #2: 15
Порожденный поток #2 - завершение
```

Рассмотрим подробнее эту программу. В ней определены три класса. Имя первого — SumArray. В нем содержится метод sumArray(), вычисляющий сумму элементов целочисленного массива. Во втором классе MyThread используется статический объект sa типа SumArray для получения суммы элементов массива. А поскольку он статический, то все экземпляры класса MyThread используют одну его копию. И наконец, в классе Sync создаются два потока, в каждом из которых должна вычисляться сумма элементов массива.

В методе sumArray() вызывается метод sleep(). Он нужен лишь для того, чтобы обеспечить переключение задач. Метод sumArray() синхронизирован, и поэтому в каждый момент времени он может использоваться только одним потоком. Следовательно, когда второй порожденный поток начинает свое выполнение, он не может вызвать метод sumArray() до тех пор, пока этот метод не завершится в первом потоке, благодаря чему обеспечивается правильность получаемого результата.

Чтобы лучше понять эффекты синхронизации, удалите ключевое слово synchronized из объявления метода sumArray(). В итоге метод sumArray() потеряет синхронизацию и может быть использован в нескольких потоках одновременно. Связанная с этим проблема заключается в том, что результат расчета суммы сохраняется в переменной sum, значение которой изменяется при каждом вызове метода sumArray() для статического объекта sa. Например, если в двух потоках одновременно сделать вызов sa.sumArray(), расчет суммы окажется неверным, поскольку в переменной sum накапливаются результаты суммирования, выполняемого одновременно в двух потоках. Ниже приведен

результат выполнения той же программы, но с удаленным ключевым словом `synchronized` в объявлении метода `sumArray()`. (Результат, полученный вами на своем компьютере, может несколько отличаться.)

```
Порожденный поток #1 - запуск
Текущее значение суммы для Порожденный поток #1: 1
Порожденный поток #2 - запуск
Текущее значение суммы для Порожденный поток #2: 1
Текущее значение суммы для Порожденный поток #1: 3
Текущее значение суммы для Порожденный поток #2: 5
Текущее значение суммы для Порожденный поток #2: 8
Текущее значение суммы для Порожденный поток #1: 11
Текущее значение суммы для Порожденный поток #2: 15
Текущее значение суммы для Порожденный поток #1: 19
Текущее значение суммы для Порожденный поток #2: 24
Сумма для Порожденный поток #2: 24
Порожденный поток #2 - завершение
Текущее значение суммы для Порожденный поток #1: 29
Сумма для Порожденный поток #1: 29

Порожденный поток #1 - завершение
```

Нетрудно заметить, что одновременные вызовы метода `sa.sumArray()` из разных потоков искажают результат.

Прежде чем переходить к рассмотрению следующей темы, перечислим основные свойства синхронизированных методов.

- Синхронизированный метод создается путем указания ключевого слова `synchronized` в его объявлении.
- Как только синхронизированный метод любого объекта получает управление, объект блокируется, и ни один синхронизированный метод этого объекта не может быть вызван другим потоком.
- Потоки, которым требуется синхронизированный метод, используемый другим потоком, ждут до тех пор, пока не будет разблокирован объект, для которого он вызывается.
- Когда синхронизированный метод завершается, объект, для которого он вызывался, разблокируется.

Синхронизированные блоки кода

Несмотря на то что создание синхронизированных методов в классах — простой и эффективный способ управления потоками, такой способ оказывается пригодным далеко не всегда. Иногда возникает потребность синхронизировать доступ к методам, в объявлении которых отсутствует ключевое слово `synchronized`. Подобная ситуация часто возникает при использовании классов, которые были созданы независимыми разработчиками и исходный код которых недоступен. В таком случае ввести в объявление нужного метода

ключевое слово synchronized вряд ли удастся. Как же тогда синхронизировать объект класса, содержащего этот метод? К счастью, данное затруднение разрешается очень просто. Достаточно ввести вызов метода в блок кода, объявленный как synchronized (синхронизированный блок).

Синхронизированный блок определяется следующим образом.

```
synchronized(ссылка_на_объект) {
   // синхронизируемые инструкции
}
```

Здесь *ссылка_на_объект* обозначает ссылку на конкретный объект, подлежащий синхронизации. Как только содержимое синхронизированного блока получит управление, ни один другой поток не сможет вызвать метод для объекта, на который указывает *ссылка_на_объект*, до тех пор пока этот блок не завершится.

Следовательно, обращение к методу sumArray() можно синхронизировать, вызвав его из синхронизированного блока. Такой способ демонстрируется в приведенной ниже переработанной версии предыдущей программы.

```
// Использование синхронизированного блока
// для управления доступом к SumArray
class SumArray {
   private int sum;

   int sumArray(int nums[]) {   ◄——— Здесь метод sumArray()
      sum = 0; // обнуление суммы       не синхронизирован

      for(int i=0; i<nums.length; i++) {
         sum += nums[i];
         System.out.println("Текущее значение суммы для " +
                            Thread.currentThread().getName() +
                            " будет " + sum);
         try {
            Thread.sleep(10); // разрешение переключения
                              // между задачами
         }
         catch(InterruptedException exc) {
            System.out.println("Поток прерван.");
         }
      }
      return sum;
   }
}

class MyThread implements Runnable {
   Thread thrd;
   static SumArray sa = new SumArray();
   int a[];
   int answer;
```

```
   // Конструктор нового потока
   MyThread(String name, int nums[]) {
      thrd = new Thread(this, name);
      a = nums;
   }

   // Создание и запуск потока с помощью фабричного метода
   public static MyThread createAndStart(String name,
                                         int nums[]) {

      MyThread myThrd = new MyThread(name, nums);

      myThrd.thrd.start(); // запуск потока
      return myThrd;
   }

   // Точка входа для потока
   public void run() {
      int sum;

      System.out.println(thrd.getName() + " - запуск.");

      // Синхронизация вызовов sumArray()
      synchronized(sa) { ◄───── Здесь вызовы метода sumArray()
         answer = sa.sumArray(a);   для объекта sa синхронизированы
      }

      System.out.println("Сумма для " + thrd.getName() +
                         " будет " + answer);
      System.out.println(thrd.getName() +
                         " - завершение.");
   }
}

class Sync {
   public static void main(String args[]) {
      int a[] = {1, 2, 3, 4, 5};

      MyThread mt1 = MyThread.createAndStart("Порожденный поток #1",
                                             a);

      MyThread mt2 = MyThread.createAndStart("Порожденный поток #2",
                                             a);
      try {
         mt1.thrd.join();
         mt2.thrd.join();
      } catch(InterruptedException exc) {
         System.out.println("Прерывание основного потока.");
      }
   }
}
```

Выполнение этой версии программы дает такой же правильный результат, как и предыдущая ее версия, в которой использовался синхронизированный метод.

Спросим у эксперта

ВОПРОС. Говорят, что существуют так называемые "утилиты параллелизма". Что это такое? И что такое Fork/Join Framework?

ОТВЕТ. Утилиты параллелизма, входящие в пакет `java.util.concurrent` (и подчиненные ему пакеты), предназначены для поддержки параллельного программирования. Среди прочего они предоставляют синхронизаторы, пулы потоков, диспетчеры исполнения и блокировки, которые расширяют возможности контроля над выполнением потока. К числу наиболее привлекательных средств прикладного интерфейса параллельного программирования относится среда Fork/Join Framework.

В Fork/Join Framework поддерживается так называемое *параллельное программирование*. Этим термином обычно обозначаются приемы программирования, использующие преимущества компьютеров с двумя и более процессорами, включая и многоядерные системы, которые позволяют разделять задачи на более мелкие подзадачи, каждая из которых выполняется на собственном процессоре. Нетрудно себе представить, что такой подход позволяет существенно повысить производительность и пропускную способность. Главное преимущество среды Fork/Join Framework заключается в простоте ее использования. Она упрощает создание многопоточных программ с автоматическим масштабированием для использования нескольких процессоров в системе. Следовательно, она облегчает разработку параллельных решений таких распространенных задач, как выполнение операций над элементами массива. Утилиты параллелизма вообще и Fork/Join Framework в частности относятся к тем средствам, которые вам стоит освоить после того, как вы приобретете определенный опыт многопоточного программирования.

Организация взаимодействия потоков с помощью методов `notify()`, `wait()` и `notifyAll()`

В качестве примера рассмотрим следующую ситуацию. Поток T, который выполняется в синхронизированном методе, нуждается в доступе к ресурсу R, который временно недоступен. Что делать потоку T? Начать выполнение цикла опросов в ожидании того момента, когда освободится ресурс R? Но тогда поток T будет связывать объект, препятствуя доступу к нему других потоков. Такое решение малопригодно, поскольку оно сводит на нет все преимущества

программирования в многопоточной среде. Будет гораздо лучше, если поток T временно разблокирует объект и позволит другим потокам воспользоваться его методами. Когда ресурс R станет доступным, поток T получит об этом уведомление и возобновит свое исполнение. Но для того чтобы такое решение можно было реализовать, необходимы средства взаимодействия потоков, с помощью которых один поток мог бы сообщить другому потоку о том, что он приостановил свое исполнение, а также получить уведомление о том, что его исполнение может быть возобновлено. Для организации подобного взаимодействия потоков в Java предусмотрены методы `wait()`, `notify()` и `notifyAll()`.

Эти методы реализованы в классе `Object`, поэтому доступны для любого объекта. Но обратиться к ним можно только из синхронизированного контекста. А применяются они следующим образом. Когда поток временно приостанавливает свое исполнение, он вызывает метод `wait()`. При этом поток переходит в состояние ожидания и монитор данного объекта освобождается, позволяя другим потокам использовать объект. Впоследствии ожидающий поток возобновит свое выполнение, когда другой поток войдет в тот же самый монитор и вызовет метод `notify()` или `notifyAll()`.

В классе `Object` определены следующие формы объявления метода `wait()`.

```
final void wait() throws InterruptedException
final void wait(long миллисекунды) throws InterruptedException
final void wait(long миллисекунды, int наносекунды) throws
   InterruptedException
```

В первой своей форме метод `wait()` переводит поток в режим ожидания до поступления уведомления. Во второй форме метода ожидание длится либо до получения уведомления, либо до тех пор, пока не истечет указанный промежуток времени. Третья форма позволяет точнее задавать период времени в наносекундах.

Ниже приведены общие формы объявления методов `notify()` и `notifyAll()`.

```
final void notify()
final void notifyAll()
```

При вызове метода `notify()` возобновляется выполнение одного ожидающего потока. Метод `notifyAll()` уведомляет все потоки об освобождении объекта, и тот поток, который имеет наивысший приоритет, получает доступ к объекту.

Прежде чем перейти к рассмотрению конкретного примера, демонстрирующего применение метода `wait()`, необходимо сделать важное замечание. Несмотря на то что метод `wait()` должен переводить поток в состояние ожидания до тех пор, пока не будет вызван метод `notify()` или `notifyAll()`, иногда поток выводится из состояния ожидания вследствие так называемой *ложной активизации*. Условия для ложной активизации слишком сложны, чтобы их можно было рассмотреть в данной книге. Достаточно лишь сказать, что компания Oracle рекомендует учитывать вероятность проявления ложной активизации и помещать вызов метода `wait()` в цикл. В этом цикле должно проверяться

условие, по которому поток переводится в состояние ожидания. Именно такой подход и применяется в приведенном ниже примере.

Пример использования методов `wait()` и `notify()`

Для того чтобы вам стала понятнее потребность в применении методов `wait()` и `notify()` в многопоточном программировании, рассмотрим пример программы, имитирующей работу часов и выводящей на экран слова `"Tick"` (тик) и `"Tock"` (так). Для этой цели создадим класс `TickTock`, который будет содержать два метода: `tick()` и `tock()`. Метод `tick()` выводит слово `"Tick"`, а метод `tock()` — слово `"Tock"`. При запуске программы, имитирующей часы, создаются два потока: в одном из них вызывается метод `tick()`, а в другом — метод `tock()`. В результате взаимодействия двух потоков на экран будет выводиться набор повторяющихся сообщений `"Tick  Tock"`, т.е. после слова `"Tick"`, обозначающего один такт, должно следовать слово `"Tock"`, обозначающее другой такт часов.

```
// Использование методов wait() и notify() для имитации часов
class TickTock {

   String state; // содержит сведения о состоянии часов

   synchronized void tick(boolean running) {
      if(!running) { // остановить часы
         state = "ticked";
         notify(); // уведомить ожидающие потоки
         return;
      }

      System.out.print("Tick ");

      state = "ticked"; // установить текущее состояние
                        // после такта "тик"
      notify(); позволить выполняться методу tock() ◄── Метод tick() посылает уведомление методу tock()
      try {
         while(!state.equals("tocked"))
            wait();// ожидать до завершения метода tock() ◄── Метод tick() ожидает завершения метода tock()
      }
      catch(InterruptedException exc) {
         System.out.println("Прерывание потока");
      }
   }

   synchronized void tock(boolean running) {
      if(!running) { // остановить часы
         state = "tocked";
         notify(); // уведомить ожидающие потоки
         return;
      }

      System.out.println("Tock");
```

```
      state = "tocked"; // установить текущее состояние
                        // после такта "так"
      notify(); // позволить выполняться методу tick() ◄─── Метод tock() посылает уведомление методу tick()
      try {
         while(!state.equals("ticked"))
            wait(); // ожидать до завершения метода tick() ◄─── Метод tock() ожидает завершения метода tick()
      }
      catch(InterruptedException exc) {
         System.out.println("Прерывание потока");
      }
   }
}

class MyThread implements Runnable {
   Thread thrd;
   TickTock ttOb;

   // Конструктор нового потока
   MyThread(String name, TickTock tt) {
      thrd = new Thread(this, name);
      ttOb = tt;
   }

   // Создание и запуск потока с помощью фабричного метода
   public static MyThread createAndStart(String name, TickTock tt) {
      MyThread myThrd = new MyThread(name, tt);

      myThrd.thrd.start(); // запуск потока
      return myThrd;
   }

   // Точка входа для потока
   public void run() {
      if(thrd.getName().compareTo("Tick") == 0) {
         for(int i=0; i<5; i++) ttOb.tick(true);
         ttOb.tick(false);
      }
      else {
         for(int i=0; i<5; i++) ttOb.tock(true);
         ttOb.tock(false);
      }
   }
}

class ThreadCom {
   public static void main(String args[]) {
      TickTock tt = new TickTock();
      MyThread mt1 = new MyThread("Tick", tt);
      MyThread mt2 = new MyThread("Tock", tt);

      try {
        mt1.thrd.join();
        mt2.thrd.join();
```

```
        } catch(InterruptedException exc) {
            System.out.println("Прерывание основного потока");
        }
    }
}
```

В результате выполнения этой программы на экране появляются следующие сообщения.

```
Tick Tock
Tick Tock
Tick Tock
Tick Tock
Tick Tock
```

Рассмотрим более подробно исходный код программы, имитирующей работу часов. В ее основу положен класс `TickTock`, в котором содержатся два взаимодействующих метода: `tick()` и `tock()`. Это взаимодействие организовано таким образом, чтобы за словом `"Tick"` всегда следовало слово `"Tock"`, затем вновь слово `"Tick"` и т.д. Обратите внимание на переменную `state`. В процессе работы имитатора часов в данной переменной хранится строка `"ticked"` или `"tocked"`, определяющая текущее состояние часов после такта "тик" или "так" соответственно. В методе `main()` создается объект `tt` типа `TickTock`, используемый для запуска двух потоков на выполнение.

Потоки создаются на основе объектов типа `MyThread`. Конструктору `MyThread()` передаются два параметра. Первый из них задает имя потока (в данном случае — `"Tick"` или `"Tock"`), а второй — ссылку на объект типа `TickTock` (в данном случае — объект `tt`). В методе `run()` из класса `MyThread` вызывается метод `tick()`, если поток называется `"Tick"`, или же метод `tock()`, если поток называется `"Tock"`. Каждый из этих методов вызывается пять раз с параметром, имеющим логическое значение `true`. Работа имитатора часов продолжается до тех пор, пока методу передается параметр с логическим значением `true`. Последний вызов каждого из методов с параметром, имеющим логическое значение `false`, останавливает имитатор работы часов.

Самая важная часть программы находится в теле методов `tick()` и `tock()` из класса `TickTock`. Начнем с метода `tick()`. Для удобства анализа ниже представлен исходный код этого метода.

```
synchronized void tick(boolean running) {
    if(!running) { // остановить часы
        state = "ticked";
        notify(); // уведомить ожидающие потоки
        return;
    }

    System.out.print("Tick ");

    state = "ticked"; // установить текущее состояние
                      // после такта "тик"
```

```
        notify(); // позволить выполняться методу tock()
        try {
            while(!state.equals("tocked"))
                wait(); // ожидать завершения метода tock()
        }
        catch(InterruptedException exc) {
            System.out.println("Прерывание потока");
        }
    }
```

Прежде всего обратите внимание на то, что в объявлении метода tick() стоит ключевое слово synchronized, указываемое в качестве модификатора доступа. Как пояснялось ранее, действие методов wait() и notify() распространяется только на синхронизированные методы. В начале метода tick() проверяется значение параметра running. Этот параметр служит для корректного завершения программы, имитирующей работу часов. Если он имеет значение false, имитатор работы часов должен быть остановлен. Если же параметр running имеет значение true, а переменная state — значение "ticked", то вызывается метод notify(), разрешающий ожидающему потоку возобновить свое исполнение. Мы еще вернемся к этому вопросу чуть позже.

По ходу работы имитируемых часов в методе tick() выводится слово "Tick", переменная state получает значение "ticked", а затем вызывается метод notify(). Вызов метода notify() возобновляет выполнение ожидающего потока. Далее в цикле while вызывается метод wait(). В итоге выполнение метода tick() будет приостановлено до тех пор, пока другой поток не вызовет метод notify(). Таким образом, очередной шаг цикла не будет выполнен до тех пор, пока другой поток не вызовет метод notify() для того же самого объекта. Поэтому, когда вызывается метод tick(), на экран выводится слово "Tick", и другой поток получает возможность продолжить свое выполнение, а затем выполнение этого метода приостанавливается.

В том цикле while, в котором вызывается метод wait(), проверяется значение переменной state. Значение "tocked", означающее завершение цикла, будет установлено только после выполнения метода tock(). Этот цикл предотвращает продолжение выполнения потока в результате ложной активизации. Если по окончании ожидания в переменной state не будет находиться значение "tocked", значит, имела место ложная активизация, и метод wait() будет вызван снова.

Метод tock() является почти точной копией метода tick(), его отличие состоит лишь в том, что он выводит на экран слово "Tock" и присваивает переменной state значение "tocked". Следовательно, когда метод tock() вызывается, он выводит на экран слово "Tock", вызывает метод notify(), а затем переходит в состояние ожидания. Если проанализировать работу сразу двух потоков, то станет ясно, что за вызовом метода tick() тотчас следует вызов метода tock(), после чего снова вызывается метод tick() и т.д. В итоге оба метода синхронизируют друг друга.

При остановке имитатора работы часов вызывается метод `notify()`. Это нужно для того, чтобы возобновить выполнение ожидающего потока. Как упоминалось выше, в обоих методах, `tick()` и `tock()`, после вывода сообщения на экран вызывается метод `wait()`. В результате при остановке имитатора работы часов один из потоков обязательно будет находиться в состоянии ожидания. Следовательно, последний вызов метода `notify()` необходим. В качестве эксперимента попробуйте удалить вызов метода `notify()` и посмотрите, что при этом произойдет. Вы увидите, что программа зависнет, и вам придется завершить ее нажатием комбинации клавиш <Ctrl+C>. Дело в том, что когда метод `tock()` в последний раз получает управление, он вызывает метод `wait()`, после чего не происходит вызов метода `notify()`, позволяющего завершиться методу `tock()`. В итоге метод `tock()` остается в состоянии бесконечного ожидания.

Если у вас еще остаются сомнения по поводу того, что методы `wait()` и `notify()` необходимы для организации нормального выполнения программы, имитирующей работу часов, замените в ее исходном коде класс `TickTock` приведенным ниже вариантом. Он отличается тем, что в нем удалены вызовы методов `wait()` и `notify()`.

```
// В этой версии вызовы методов wait() и notify() отсутствуют
class TickTock {

   String state; // содержит сведения о состоянии часов

   synchronized void tick(boolean running) {
     if(!running) { // остановить часы
        state = "ticked";
        return;
     }

      System.out.print("Tick ");

      state = "ticked"; // установить текущее состояние
                        // после такта "тик"
   }

   synchronized void tock(boolean running) {
      if(!running) { // остановить часы
         state = "tocked";
         return;
      }

      System.out.println("Tock");

      state = "tocked"; // установить текущее состояние
                        // после такта "так"
   }
}
```

Теперь программа выводит на экран следующие сообщения.

```
Tick Tick Tick Tick Tick Tock
Tock
Tock
Tock
Tock
```

Причина подобного поведения заключается в том, что методы `tick()` и `tock()` не взаимодействуют друг с другом.

Спросим у эксперта

ВОПРОС. Мне приходилось слышать, как при обсуждении многопоточных программ, ведущих себя не так, как ожидалось, использовался термин *взаимоблокировка*. Что это такое и как этого избежать? Кроме того, что такое *состояние гонки* и как с этим бороться?

ОТВЕТ. Взаимоблокировка возникает в тех случаях, когда один поток ожидает завершения некоторых действий другим потоком, а другой — того же самого от первого. В итоге оба потока оказываются в состоянии ожидания, и их выполнение не может возобновиться. Это можно сравнить с поведением двух джентльменов, каждый из которых ни за что не соглашается пройти в дверь раньше другого.

На первый взгляд, предотвратить взаимоблокировку нетрудно, но на самом деле это совсем не так. Например, взаимоблокировка может возникнуть косвенным образом. Причины ее не всегда понятны, поскольку между потоками нередко организуется довольно сложное взаимодействие. Единственный способ предотвратить взаимоблокировку — тщательно проектировать и тестировать создаваемый исходный код. Если многопоточная программа время от времени зависает, то, скорее всего, имеет место взаимоблокировка.

Состояние гонки возникает в тех случаях, когда несколько потоков пытаются одновременно получить доступ к общему ресурсу без должной синхронизации. Так, в одном потоке может сохраняться значение в переменной, а в другом — инкрементироваться текущее значение этой же переменной. В отсутствие синхронизации конечный результат будет зависеть от того, в каком именно порядке выполняются потоки: инкрементируется ли значение переменной во втором потоке или же оно сохраняется в первом. В подобных ситуациях конечный результат зависит от того, какой из потоков завершится первым. Возникающее состояние гонки, как и взаимоблокировку, непросто обнаружить. Поэтому его лучше предотвратить, синхронизируя должным образом доступ к общим ресурсам на стадии программирования.

Приостановка, возобновление и остановка потоков

Иногда бывает полезно приостановить или даже полностью прекратить выполнение потока. Допустим, отдельный поток используется для отображения времени. Если пользователю не нужны часы на экране, то отображающий их поток можно приостановить. Независимо от причин, по которым требуется временная остановка потока, сделать это нетрудно, как, впрочем, и возобновить выполнение потока.

Механизмы приостановки, возобновления и остановки потоков менялись в разных версиях Java. До появления версии Java 2 для этих целей использовались методы `suspend()`, `resume()` и `stop()`, определенные в классе `Thread`. Ниже приведены общие формы их объявления.

```
final void resume()
final void suspend()
final void stop()
```

На первый взгляд кажется, что упомянутые выше методы удобны для управления потоками, но пользоваться ими все же не рекомендуется по следующим причинам. При выполнении метода `suspend()` иногда возникают серьезные осложнения, приводящие к взаимоблокировке. Метод `resume()` сам по себе безопасен, но применяется только в сочетании с методом `suspend()`. Что же касается метода `stop()` из класса `Thread`, то и он не рекомендуется к применению начиная с версии Java 2, поскольку может вызывать порой серьезные осложнения в работе многопоточных программ.

Если методы `suspend()`, `resume()` и `stop()` нельзя использовать для управления потоками, то может показаться, что приостановить, возобновить и остановить поток вообще нельзя. Но, к счастью, это не так. Поток следует разрабатывать таким образом, чтобы в методе `run()` периодически осуществлялась проверка того, следует ли приостановить, возобновить или остановить поток. Обычно для этой цели используются две флаговые переменные: одна — для приостановки и возобновления потока, другая — для остановки потока. Если флаговая переменная, управляющая приостановкой потока, установлена в состояние исполнения, то метод `run()` должен обеспечить продолжение выполнения потока. Если же эта флаговая переменная находится в состоянии приостановки, значит, в работе потока должна наступить пауза. А если переменная, управляющая остановкой потока, находится в состоянии остановки, то выполнение потока должно прекратиться.

Следующий пример программы демонстрирует один из способов реализации собственных версий методов `suspend()`, `resume()` и `stop()`.

```
// Приостановка, возобновление и остановка потока

class MyThread implements Runnable {
   Thread thrd;
```

```
boolean suspended;  ◄─── Приостанавливает поток при значении true
boolean stopped;  ◄─── Останавливает поток при значении true

MyThread(String name) {
   thrd = new Thread(this, name);
   suspended = false;
   stopped = false;
}

// Создание и запуск потока с помощью фабричного метода
public static MyThread createAndStart(String name) {
   MyThread myThrd = new MyThread(name);

   myThrd.thrd.start(); // запуск потока
   return myThrd;
}

// Точка входа для потока
public void run() {
   System.out.println(thrd.getName() + " - запуск.");
   try {
      for(int i = 1; i < 1000; i++) {
         System.out.print(i + " ");
         if((i%10)==0) {
            System.out.println();
            Thread.sleep(250);
         }

         // Использование синхронизированного блока для
         // проверки значения переменных suspended и stopped
         synchronized(this) {  ◄───┐
            while(suspended) {     │
               wait();          Этот синхронизированный блок
            }                   используется для тестирования
            if(stopped) break;  переменных suspended и stopped
         }
      }
   } catch (InterruptedException exc) {
      System.out.println(thrd.getName() + " - прерван.");
   }
   System.out.println(thrd.getName() + " - выход.");
}

// Остановить поток
synchronized void mystop() {
   stopped = true;

   // Следующие инструкции полностью останавливают
   // приостановленный поток
   suspended = false;
   notify();
}
```

```
    // Приостановить поток
    synchronized void mysuspend() {
        suspended = true;
    }

    // Возобновить поток
    synchronized void myresume() {
        suspended = false;
        notify();
    }
}

class Suspend {
    public static void main(String args[]) {
        MyThread mt1 = MyThread.createAndStart("Мой поток");

        try {
            Thread.sleep(1000); // позволить потоку ob1 начать
                                // выполнение
            mt1.mysuspend();
            System.out.println("Приостановка потока.");
            Thread.sleep(1000);

            mt1.myresume();
            System.out.println("Возобновление потока.");
            Thread.sleep(1000);

            mt1.mysuspend();
            System.out.println("Приостановка потока.");
            Thread.sleep(1000);

            mt1.myresume();
            System.out.println("Возобновление потока.");
            Thread.sleep(1000);

            mt1.mysuspend();
            System.out.println("Остановка потока.");
            mt1.mystop();
        } catch (InterruptedException e) {
            System.out.println("Прерывание основного потока ");
        }

        // Ожидание завершения потока
        try {
            mt1.thrd.join();
        } catch (InterruptedException e) {
            System.out.println("Прерывание основного потока ");
        }
        System.out.println("Выход из основного потока.");
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Мой поток - запуск
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
Приостановка потока
Возобновление потока
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
Приостановка потока
Возобновление потока
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100
101 102 103 104 105 106 107 108 109 110
111 112 113 114 115 116 117 118 119 120
Остановка потока
Мой поток - выход
Выход из основного потока
```

Эта программа работает следующим образом. В классе потока `MyThread` определены две логические переменные, `suspended` и `stopped`, управляющие временной и полной остановкой потока. В конструкторе этого класса обеим переменным присваивается значение `false`. Метод `run()` содержит синхронизированный блок, в котором проверяется состояние переменной `suspended`. Если эта переменная имеет значение `true`, вызывается метод `wait()`, приостанавливающий выполнение потока. Значение `true` присваивается переменной `suspended` в методе `mysuspend()`, и поэтому данный метод следует вызвать для приостановки потока. Для возобновления потока служит метод `myresume()`, в котором переменной `suspended` присваивается значение `false` и вызывается метод `notify()`.

Для остановки потока следует вызвать метод `mystop()`, в котором переменной `stopped` присваивается значение `true`. Кроме того, в методе `mystop()` переменной `suspended` присваивается значение `false` и вызывается метод `notify()`. Это необходимо для прекращения работы потока, выполнение которого ранее было приостановлено.

Спросим у эксперта

ВОПРОС. Многопоточное программирование, по-видимому, является важным средством повышения производительности программ. Какие рекомендации можно дать по его эффективному применению?

ОТВЕТ. Самое главное для эффективного многопоточного программирования — мыслить категориями параллельного, а не последовательного выполнения

кода. Так, если в одной программе имеются две подсистемы, которые могут работать параллельно, то их следует организовать в отдельные потоки. Но делать это следует очень внимательно и тщательно, поскольку слишком большое количество потоков приводит не к повышению, а к снижению производительности. Следует также учитывать дополнительные издержки, связанные с переключением контекста. Так, если создать слишком много потоков, то на смену контекста уйдет больше времени ЦП, чем на выполнение самой программы!

Упражнение 11.2 Применение основного потока

`UseMain.java`

В каждой программе на Java имеется хотя бы один поток, называемый *основным*. Этот поток автоматически получает управление при запуске программы на выполнение. В этом упражнении будет продемонстрировано, что основным потоком можно управлять точно так же, как и любым другим. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл `UseMain.java`.
2. Для доступа к основному потоку нужно получить ссылающийся на него объект типа `Thread`. Для этого следует вызвать метод `currentThread()`, являющийся статическим членом класса `Thread`. Ниже приведено объявление этого метода.

   ```
   static Thread currentThread()
   ```

   Метод `currentThread()` возвращает ссылку на тот поток, из которого он вызывается. Так, если вызвать метод `currentThread()` из основного потока, то можно получить ссылку на этот поток. А имея ссылку на основной поток, можно управлять им.
3. Введите в файл, приведенный ниже, исходный код программы. В процессе ее выполнения сначала извлекается ссылка на основной поток, а затем определяются и устанавливаются имя и приоритет потока.

```
/*
   Упражнение 11.2

   Управление основным потоком
*/

class UseMain {
   public static void main(String args[]) {
      Thread thrd;
```

```
        // Получить основной поток
        thrd = Thread.currentThread();

        // Отобразить имя основного потока
        System.out.println("Имя основного потока: " +
                           thrd.getName());

        // Отобразить приоритет основного потока
        System.out.println("Приоритет: " +
                           thrd.getPriority());

        System.out.println();

        // Установить имя и приоритет основного потока
        System.out.println("Установка имени и приоритета\n");
        thrd.setName("Поток #1");
        thrd.setPriority(Thread.NORM_PRIORITY+3);

        System.out.println("Новое имя основного потока: " +
                           thrd.getName());

        System.out.println("Новое значение приоритета: " +
                           thrd.getPriority());
    }
}
```

4. Ниже приведен результат выполнения данной программы.

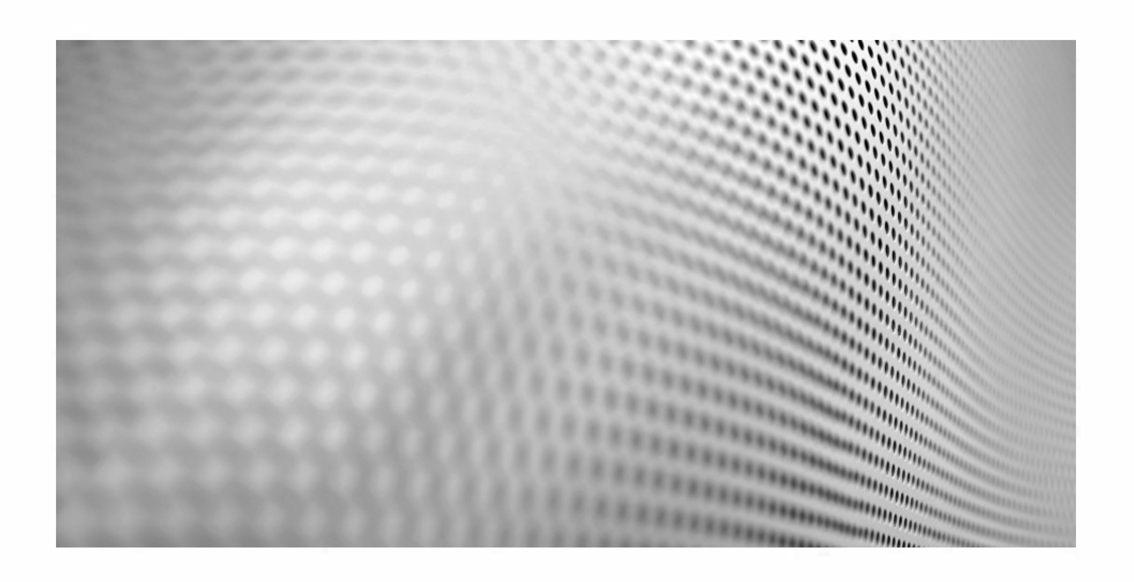
```
Имя основного потока: main
Приоритет: 5

Установка имени и приоритета

Новое имя основного потока: Поток #1
Новое значение приоритета: 8
```

5. Выполняя операции над основным потоком, необходимо соблюдать осторожность. Так, если добавить в конце метода `main()` приведенный ниже код, программа никогда не завершится, потому что будет ожидать завершения основного потока.

```
try {
   thrd.join();
} catch(InterruptedException exc) {
   System.out.println("Прервано выполнение потока.");
}
```

Вопросы и упражнения для самопроверки

1. Каким образом имеющиеся в Java средства многопоточного программирования обеспечивают создание более эффективных программ?
2. Для поддержки многопоточного программирования в Java предусмотрены класс __________ и интерфейс ___________.
3. В каких случаях при создании выполняемого объекта следует отдать предпочтение расширению класса `Thread`, а не реализации интерфейса `Runnable`?
4. Покажите, как с помощью метода `join()` можно организовать ожидание завершения потокового объекта `MyThrd`.
5. Покажите, как установить приоритет потока `MyThrd` на три уровня выше нормального приоритета.
6. Что произойдет, если в объявлении метода указать ключевое слово `synchronized`?
7. Методы `wait()` и `notify()` предназначены для обеспечения __________ ________________.
8. Внесите в класс `TickTock` изменения для организации фактического отсчета времени. Первую половину секунды должен занимать вывод на экран слова `"Tick"`, а вторую — вывод слова `"Tock"`. Таким образом, сообщение `"Tick-Tock"` должно соответствовать одной секунде отсчитываемого времени. (Время переключения контекстов можно не учитывать.)
9. Почему в новых программах на Java не следует применять методы `suspend()`, `resume()` и `stop()`?
10. С помощью какого метода из класса `Thread` можно получить имя потока?
11. Какое значение возвращает метод `isAlive()`?
12. Попытайтесь самостоятельно реализовать средства синхронизации в классе `Queue`, разработанном в предыдущих главах. Ваша цель — обеспечить корректное функционирование класса в условиях многопоточной обработки.



Глава 12

Перечисления, автоупаковка, статический импорт и аннотации

В этой главе...

- Основные сведения о перечислимых типах
- Объектные свойства перечислений
- Применение методов `values()` и `valueof()` к перечислениям
- Создание перечислений с конструкторами, переменными экземпляров и методами
- Применение методов `ordinal()` и `compareTo()`, наследуемых перечислениями от класса `Enum`
- Использование объектных оболочек Java
- Основные сведения об автоупаковке и автораспаковке
- Использование автоупаковки в методах
- Использование автоупаковки в выражениях
- Использование статического импорта
- Общий обзор аннотаций

В данной главе рассматриваются перечисления, аннотации, механизмы автоупаковки и статического импорта. И хотя ни одно из этих средств первоначально не входило в Java (все они появились в версии JDK 5), каждое из них увеличивает мощь языка и делает его использование более удобным. Включение в язык Java перечислений и автоупаковки удовлетворило давнюю потребность программистов в этих средствах, статический импорт упростил использование статических членов классов, а введение аннотаций позволило внедрять в исходные файлы дополнительную информацию. Совокупность этих средств обеспечила более эффективные способы решения часто встречающихся задач программирования. В этой главе также рассматриваются оболочки типов Java.

Перечисления

В своей простейшей форме *перечисление* — это список именованных констант, определяющих новый тип данных. В объектах перечислимого типа могут храниться лишь значения, содержащиеся в этом списке. Таким образом, перечисления позволяют определять новый тип данных, характеризующийся строго определенным рядом допустимых значений.

Перечисления часто встречаются в повседневной жизни. В качестве примера можно привести номинальные значения денежных купюр, а также названия дней недели или месяцев в году — все это перечисления.

С точки зрения программирования перечисления оказываются удобными в тех случаях, когда нужно определить набор значений, представляющих коллекцию элементов. Например, с помощью перечисления можно представить набор кодов состояния (успешное завершение, ожидание, ошибка, необходимость повторной попытки), которые соответствуют различным стадиям какого-либо процесса. Разумеется, для этих целей вполне можно использовать константы типа `final`, но перечисления обеспечивают более структурированный подход к решению подобных задач.

Основные сведения о перечислениях

Перечисления создаются с использованием ключевого слова `enum`. Вот так, например, может выглядеть простое перечисление, представляющее различные виды транспортных средств.

```
// Перечисление, представляющее разновидности транспортных средств
enum Transport {
   CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
```

Идентификаторы `CAR`, `TRUCK` и р. — это *константы перечисления*. Каждый из них неявно объявлен как открытый (`public`), статический (`static`) член перечисления `Transport`. Типом этих констант является тип перечисления (в данном случае `Transport`). В Java подобные константы называют *самотипизированными*, где приставка "само" относится к перечислению, к которому они принадлежат.

Определив перечисление, можно создавать переменные этого типа. Однако, несмотря на то что перечисление — это тип класса, объекты этого класса создаются без применения оператора `new`. Переменные перечислимого типа создаются подобно переменным элементарных типов. Например, для определения переменной `tp` типа `Transport` понадобится следующая строка кода:

```
Transport tp;
```

Поскольку переменная `tp` относится к типу `Transport`, ей можно присваивать только те значения, которые определены для данного типа. Например, в следующей строке кода переменной `tp` присваивается значение `AIRPLANE`:

```
tp = Transport.AIRPLANE;
```

Обратите внимание на то, что значению `AIRPLANE` предшествует указанный через точку тип `Transport`.

Для проверки равенства констант перечислимого типа используется оператор сравнения (==). Например, в следующей строке кода содержимое переменной `tp` сравнивается с константой `TRAIN`:

```
if(tp == Transport.TRAIN) // ...
```

Перечисления можно использовать в качестве селектора в переключателе `switch`. Разумеется, в ветвях `case` должны указываться только константы того

же перечислимого типа, что и в выражении switch. Например, вполне допустим код наподобие следующего.

```
// Использование перечисления для управления инструкцией switch
switch(tp) {
   case CAR:
      // ...
   case TRUCK:
      // ...
```

Заметьте, что в ветвях case инструкции switch используются простые имена констант, а не уточненные. Так, в приведенном выше коде вместо полного имени Transport.TRUCK используется простое имя TRUCK. Этого достаточно, поскольку тип перечисления в выражении инструкции switch неявно задает тип констант в ветвях case. Более того, если вы попытаетесь указать тип констант явным образом, компилятор выдаст сообщение об ошибке.

При отображении константы перечислимого типа, например с помощью метода println(), выводится ее имя. Так, в результате выполнения следующей инструкции отобразится имя BOAT:

```
System.out.println(Transport.BOAT);
```

Ниже приведен пример программы, демонстрирующий все особенности применения перечисления Transport.

```
// Использование перечисления Transport.

// Перечисление, представляющее разновидности транспортных средств
enum Transport {
   CAR, TRUCK, AIRPLANE, TRAIN, BOAT ◄——— Объявление перечисления
}

class EnumDemo {
   public static void main(String args[])
   {
      Transport tp; ◄——— Объявление ссылки Transport

      tp = Transport.AIRPLANE;◄——— Присваивание переменной tp константы AIRPLANE

      // Отобразить перечислимое значение
      System.out.println("Значение tp: " + tp);
      System.out.println();

      tp = Transport.TRAIN;

      // Сравнить два перечислимых значения
      if(tp == Transport.TRAIN) ◄——— Сравнение двух объектов Transport на предмет равенства
         System.out.println("tp содержит TRAIN\n");

      // Использование перечисления для управления
      // инструкцией switch
      switch(tp) { ◄——— Использование перечисления для управления инструкцией switch
```

```
         case CAR:
            System.out.println("Автомобиль везет людей");
            break;
         case TRUCK:
            System.out.println("Грузовик перевозит груз");
            break;
         case AIRPLANE:
            System.out.println("Самолет летит");
            break;
         case TRAIN:
            System.out.println("Поезд движется по рельсам");
            break;
         case BOAT:
            System.out.println("Лодка плывет по реке");
            break;
      }
   }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
Значение tp: AIRPLANE

tp содержит TRAIN

Поезд движется по рельсам
```

Прежде чем вигаться дальше, следует сделать одно замечание. Имена констант в перечислении `Transport` указываются прописными буквами (например, одна из констант перечисления называется `CAR`, а не `car`). Однако это требование не является обязательным. Никаких особых требований к регистру символов в именах констант не предъявляется. Но поскольку константы перечислимого типа обычно играют ту же роль, что и финальные (`final`) переменные, которые традиционно обозначаются прописными буквами, для записи имен констант принято использовать тот же способ. И хотя на этот счет существуют различные точки зрения, в примерах программ, представленных в книге, для констант перечислимого типа будут использоваться имена, записанные прописными буквами.

Перечисления Java являются типами классов

Несмотря на то что предыдущие примеры позволили продемонстрировать создание и использование перечислений, они не дают полного представления обо всех возможностях этого типа данных. В Java, в отличие от других языков программирования, *перечисления реализованы как типы классов*. И хотя для создания экземпляров класса `enum` не требуется использовать оператор `new`, во всех остальных отношениях они ничем не отличаются от классов. Реализация перечислений Java в виде классов позволила значительно расширить их возможности. В частности, допускается определение конструкторов перечислений,

добавление в них объектных переменных и методов и даже создание перечислений, реализующих интерфейсы.

Методы values() и valueOf()

Все перечисления автоматически включают два предопределенных метода, `values()` и `valueOf()`, общие формы объявления которых приведены ниже.

```
public static перечислимый_тип[] values()
public static перечислимый_тип valueOf(String str)
```

Метод `values()` возвращает массив, содержащий список констант перечисления, а метод `valueOf()` — константу перечисления, значение которой соответствует строке *str*, переданной методу в качестве аргумента. В обоих случаях *перечислимый_тип* — это тип перечисления. Например, в случае рассмотренного выше перечисления `Transport` вызов метода `Transport.valueOf("TRAIN")` вернет значение `TRAIN` типа `Transport`. Рассмотрим пример программы, демонстрирующей использование методов `values()` и `valueOf()`.

```
// Использование встроенных методов перечислений.

// Перечисление, представляющее разновидности транспортных средств
enum Transport {
   CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}

class EnumDemo2 {
   public static void main(String args[])
   {
      Transport tp;

      System.out.println("Константы Transport:");

      // применение метода values()
      Transport allTransports[] = Transport.values(); ◄── Получение массива констант Transport
      for(Transport t : allTransports)
         System.out.println(t);

      System.out.println();

      // применение метода valueOf()
      tp = Transport.valueOf("AIRPLANE"); ◄── Получение константы AIRPLANE
      System.out.println("tp содержит " + tp);

   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Константы Transport:
CAR
TRUCK
```

```
AIRPLANE
TRAIN
BOAT

tp содержит AIRPLANE
```

Обратите внимание на то, что в данном примере для перебора массива констант, полученного с помощью метода `values()`, используется цикл типа `for-each`. Чтобы сделать пример более наглядным, в нем создается переменная `allTransports`, которой присваивается ссылка на массив констант перечисления. Однако делать это вовсе не обязательно, и цикл `for` можно переписать так, как показано ниже. (В этом случае необходимость в использовании дополнительной переменной `allTransports` отпадает.)

```
for(Transport t : Transport.values())
   System.out.println(t);
```

Обратите внимание также на то, что значение, соответствующее имени `AIRPLANE`, было получено в результате вызова метода `valueOf()`:

```
tp = Transport.valueOf("AIRPLANE");
```

Как объяснялось ранее, метод `valueOf()` возвращает значение перечислимого типа, которое ассоциировано с именем константы, представленной в виде строки.

Конструкторы, методы, переменные экземпляра и перечисления

Очень важно, чтобы вы понимали, что в перечислении каждая константа является объектом класса данного перечисления. Таким образом, перечисление может иметь конструкторы, методы и переменные экземпляра. Если определить для объекта перечислимого типа конструктор, он будет вызываться всякий раз при создании константы перечисления. Для каждой константы перечислимого типа можно вызвать любой метод, определенный в перечислении. Кроме того, у каждой константы перечислимого типа имеется собственная копия любой переменной экземпляра, определенной в перечислении. Ниже приведена переработанная версия предыдущей программы, которая демонстрирует использование конструктора, переменной экземпляра, а также метода перечисления `Transport` и выводит для каждого вида транспортного средства его типичную скорость движения.

```
// Использование конструктора, переменной экземпляра и
// метода перечисления
enum Transport {
   CAR(100), TRUCK(80), AIRPLANE(900), TRAIN(120), BOAT(35); ◄── Обратите внимание на значения инициализации

   private int speed; // типичная скорость транспортного средства ◄── Добавить переменную экземпляра
```

```
   // конструктор
   Transport(int s) { speed = s; } ◄——— Добавить конструктор

   // метод
   int getSpeed() { return speed; }◄——— Добавить метод
}

class EnumDemo3 {
   public static void main(String args[])
   {
      Transport tp;

      // Отобразить скорость самолета
      System.out.println("Типичная скорость самолета: " +
                         Transport.AIRPLANE.getSpeed() +  ◄—┐
                         " км в час\n");      Получение значения скорости
                                              путем вызова метода getSpeed()

      // Отобразить все виды транспорта и скорости их движения
      System.out.println("Типичные скорости движения
                          транспортных средств");
      for(Transport t : Transport.values())
         System.out.println(t + ": " + t.getSpeed() + " км в час");
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Типичная скорость самолета: 900 км в час

Типичные скорости движения транспортных средств
CAR: 100 км в час
TRUCK: 80 км в час
AIRPLANE: 900 км в час
TRAIN: 120 км в час
BOAT: 35 км в час
```

В этой версии программы перечисление `Transport` претерпело ряд изменений. Во-первых, появилась переменная экземпляра `speed`, используемая для хранения скорости движения транспортного средства. Во-вторых, в перечисление `Transport` добавлен конструктор, которому передается значение скорости. И в-третьих, в перечисление добавлен метод `getSpeed()`, возвращающий значение переменной `speed`, т.е. скорость движения данного транспортного средства.

Когда переменная `tp` объявляется в методе `main()`, для каждой константы перечисления автоматически вызывается конструктор `Transport()`. Аргументы, передаваемые конструктору, указываются в скобках после имени константы, как показано ниже:

```
CAR(100), TRUCK(80), AIRPLANE(900), TRAIN(120), BOAT(35);
```

Числовые значения, передаваемые конструктору `Transport()` через параметр `s`, присваиваются переменной `speed`. Обратите внимание на то, что список констант перечислимого типа завершается точкой с запятой. Последней в этом списке указана константа `BOAT`. Точка с запятой требуется в том случае, когда класс перечисления содержит наряду с константами и другие члены.

У каждой константы перечислимого типа имеется собственная копия переменной `speed`, что позволяет получить скорость передвижения конкретного транспортного средства, вызвав метод `getSpeed()`. Например, в методе `main()` скорость самолета определяется с помощью следующего вызова:

```
Transport.AIRPLANE.getSpeed()
```

Скорость каждого транспортного средства определяется в процессе перебора констант перечислимого типа в цикле `for`. А поскольку каждая такая константа имеет собственную копию переменной `speed`, то значения скорости, ассоциированные с разными константами, отличаются друг от друга. Такой принцип организации перечислений довольно эффективен, но он возможен только в том случае, если перечисления реализованы в виде классов, как это сделано в Java.

В предыдущем примере использовался только один конструктор, но перечисления, как и обычные классы, допускают любое число конструкторов.

Спросим у эксперта

ВОПРОС. Теперь, когда в Java включены перечисления, можно ли считать, что финальные (`final`) переменные больше не нужны?

ОТВЕТ. Нет, нельзя. Перечисления удобны в тех случаях, когда приходится иметь дело со списками элементов, которые должны представляться идентификаторами. В то же время финальные переменные целесообразно применять для хранения постоянных значений, например размеров массивов, которые многократно используются в программе. Таким образом, у каждого из этих языковых средств имеется своя область применения. Преимущество перечислений проявляется в тех случаях, когда переменные типа `final` не совсем удобны.

Два важных ограничения

В отношении перечислений действуют два ограничения. Во-первых, перечисление не может быть подклассом другого класса. И во-вторых, перечисление не может выступать в качестве суперкласса. Иными словами, перечислимый тип `enum` нельзя расширять. Если бы это было не так, перечисления вели бы себя как обычные классы. Основной же особенностью перечислений является создание констант в виде объектов того класса, в котором они определены.

Перечисления наследуются от класса Enum

Несмотря на то что перечисление не может наследовать суперкласс, все перечисления автоматически наследуют переменные и методы класса `java.lang.Enum`. В этом классе определен ряд методов, доступных всем перечислениям. И хотя большинство этих методов используются редко, тем не менее два из них иногда применяются в программах на Java. Это методы `ordinal()` и `compareTo()`.

Метод `ordinal()` позволяет получить так называемое *порядковое значение*, которое указывает позицию константы в списке констант перечисления. Ниже приведена общая форма объявления метода `ordinal()`:

```
final int ordinal()
```

Этот метод возвращает порядковое значение вызывающей константы. Отсчет порядковых значений начинается с нуля. Следовательно, в перечислении `Transport` порядковое значение константы `CAR` равно нулю, константы `TRUCK` — 1, константы `AIRPLANE` — 2 и т.д.

Для сравнения порядковых значений двух констант одного и того же перечисления можно воспользоваться методом `compareTo()`. Ниже приведена общая форма объявления этого метода:

```
final int compareTo(перечислимый_тип e)
```

Здесь *перечислимый_тип* — это тип перечисления, а *e* — константа, сравниваемая с вызывающей константой. При этом не следует забывать, что вызывающая константа и константа *e* должны относиться к одному и тому же перечислимому типу. Если порядковое значение вызывающей константы меньше порядкового значения константы *e*, то метод `compareTo()` возвращает отрицательное значение. Если же их порядковые значения совпадают, возвращается нулевое значение. И наконец, если порядковое значение вызывающей константы больше порядкового значения константы *e*, метод возвращает положительное значение.

Ниже приведен пример программы, демонстрирующий применение методов `ordinal()` и `compareTo()`.

```
// Использование методов ordinal() и compareTo().

// Перечисление, представляющее разновидности транспортных средств
enum Transport {
   CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}

class EnumDemo4 {
   public static void main(String args[])
   {
      Transport tp, tp2, tp3;
```

```
        // Получить порядковые значения с помощью метода ordinal()
        System.out.println("Константы перечисления Transport и их
                            порядковые значения: ");
        for(Transport t : Transport.values())
           System.out.println(t + " " + t.ordinal()); ◄── Получение порядковых
                                                            значений
        tp = Transport.AIRPLANE;
        tp2 = Transport.TRAIN;
        tp3 = Transport.AIRPLANE;

        System.out.println();
                                                 Сравнение порядковых значений
        // Демонстрация использования метода compareTo()
        if(tp.compareTo(tp2) < 0) ◄──────────────────────────────┘
           System.out.println(tp + " идет перед " + tp2);

        if(tp.compareTo(tp2) > 0)
           System.out.println(tp2 + " идет перед " + tp);

        if(tp.compareTo(tp3) == 0)
           System.out.println(tp + " совпадает с " + tp3);
    }
}
```

Результат выполнения данной программы выглядит следующим образом.

```
Константы перечисления Transport и их
порядковые значения:
CAR 0
TRUCK 1
AIRPLANE 2
TRAIN 3
BOAT 4

AIRPLANE идет перед TRAIN
AIRPLANE совпадает с AIRPLANE
```

Упражнение 12.1 Автоматизированный светофор

`TrafficLightDemo.java`

Перечисления особенно удобны в тех случаях, когда в программу необходимо включить набор констант, конкретные значения которых неважны, — достаточно, чтобы они отличались друг от друга. Необходимость в подобных константах часто возникает при написании программ. В качестве показательного примера можно привести обработку ряда фиксированных состояний некоего устройства. Допустим, требуется написать код, управляющий светофором, трем состояниям которого соответствуют зеленый, желтый и красный цвет. Этот код должен периодически переключать светофор из одного состояния в другое. Кроме того, данный код должен передавать некоему другому коду информацию о текущем цвете светофора и предоставлять ему возможность задавать нужный начальный цвет. Отсюда следует, что необходимо каким-то образом представить три состояния

светофора. И хотя для этого вполне можно было бы использовать целочисленные значения, например, 1, 2 и 3, или символьные строки red (красный), green (зеленый) и yellow (желтый), лучше воспользоваться перечислением. С помощью перечисления можно написать более эффективный и структурированный код, чем тот, в котором применяются символьные строки или целочисленные значения.

В этом упражнении нам предстоит сымитировать автоматизированный светофор. Наряду с использованием перечислений в нем будет дополнительно продемонстрирован еще один пример организации многопоточной обработки и синхронизации потоков. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл TrafficLightDemo.java.
2. Начните с создания перечисления TrafficLightColor, представляющего три состояния светофора.

```
// Перечисление, представляющее цвета светофора
enum TrafficLightColor {
   RED, GREEN, YELLOW
}
```

   Каждая из констант в этом перечислении соответствует определенному цвету светофора.
3. Далее начните определять класс TrafficLightSimulator, как показано ниже. Этот класс инкапсулирует имитацию светофора.

```
// Автоматизированное управление светофором
class TrafficLightSimulator implements Runnable {
   private TrafficLightColor tlc; // текущий цвет светофора
   boolean stop = false; // для остановки имитации установить в true
   boolean changed = false; // true, если светофор переключился

   TrafficLightSimulator(TrafficLightColor init) {
      tlc = init;
   }

   TrafficLightSimulator() {
      tlc = TrafficLightColor.RED;
   }
```

   Заметьте, что класс TrafficLightSimulator реализует интерфейс Runnable. Это необходимо, поскольку для переключения цветов светофора будет использоваться отдельный поток. Для класса TrafficLightSimulator определены два конструктора. Первый из них позволяет задать начальный цвет светофора, второй устанавливает для светофора красный цвет по умолчанию.

   Далее рассмотрим переменные экземпляра. Ссылка на поток, регулирующий состояние светофора, хранится в переменной thrd. Информация о

текущем цвете хранится в переменной `tlc`. Переменная `stop` служит для остановки имитации. Первоначально она имеет значение `false`. Имитация светофора будет действовать до тех пор, пока эта переменная не примет логическое значение `true`. И наконец, переменная `changed` получает значение `true` при переключении светофора, когда его цвет меняется.

4. Добавьте приведенный ниже метод `run()`, запускающий имитацию автоматизированного светофора.

```
// Запуск имитации автоматизированного светофора
public void run() {
   while(!stop) {
      try {
         switch(tlc) {
            case GREEN:
               Thread.sleep(10000); // зеленый на 10 секунд
               break;
            case YELLOW:
               Thread.sleep(2000);  // желтый на 2 секунды
               break;
            case RED:
               Thread.sleep(12000); // красный на 12 секунд
               break;
         }
      } catch(InterruptedException exc) {
         System.out.println(exc);
      }
      changeColor();
   }
}
```

Этот метод циклически переключает цвета светофора. Сначала выполнение потока приостанавливается на заданный промежуток времени, который выбирается в зависимости от конкретного цвета светофора. Затем вызывается метод `changeColor()`, переключающий цвет светофора.

5. Добавьте приведенный ниже метод `changeColor()`, переключающий цвет светофора.

```
// Переключение цвета светофора
synchronized void changeColor() {
   switch(tlc) {
      case RED:
         tlc = TrafficLightColor.GREEN;
         break;
      case YELLOW:
         tlc = TrafficLightColor.RED;
         break;
      case GREEN:
         tlc = TrafficLightColor.YELLOW;
   }
```

```
        changed = true;
        notify(); // уведомить о переключении цвета светофора
    }
```

В инструкции switch проверяется информация о цвете светофора, хранящаяся в переменной tlc, после чего этой переменной присваивается другой цвет. Обратите внимание на то, что этот метод синхронизирован. Это необходимо было сделать потому, что он вызывает метод notify(), уведомляющий о смене цвета. (Напомним, что обратиться к методу notify() можно только из синхронизированного контекста.)

6. Добавьте метод waitForChange(), ожидающий переключения цвета светофора.

```
// Ожидание переключения цвета светофора
synchronized void waitForChange() {
    try {
        while(!changed)
            wait(); // ожидать переключения цвета светофора
        changed = false;
    } catch(InterruptedException exc) {
        System.out.println(exc);
    }
}
```

Действие этого метода ограничивается вызовом метода wait(). Возврат из него не произойдет до тех пор, пока в методе changeColor() не будет вызван метод notify(). Следовательно, метод waitForChange() не завершится до переключения цвета светофора.

7. Добавьте метод getColor(), возвращающий текущий цвет светофора, а вслед за ним — метод cancel(), останавливающий имитацию светофора, присваивая переменной stop значение true. Ниже приведен исходный код обоих методов.

```
// Возврат текущего цвета
synchronized TrafficLightColor getColor() {
    return tlc;
}

// Прекращение имитации светофора
synchronized void cancel() {
    stop = true;
}
```

8. Ниже приведен полный исходный код программы, имитирующей автоматизированный светофор с помощью перечисления.

```
// Упражнение 12.1

// Имитация автоматизированного светофора с использованием
// перечисления.
```

```
// Перечисление, представляющее цвета светофора
enum TrafficLightColor {
   RED, GREEN, YELLOW
}

// Имитация автоматизированного светофора
class TrafficLightSimulator implements Runnable {
   private TrafficLightColor tlc; // текущий цвет светофора
   boolean stop = false; // для остановка имитации установить в true
   boolean changed = false; // true, если светофор переключился

   TrafficLightSimulator(TrafficLightColor init) {
      tlc = init;
   }

   TrafficLightSimulator() {
      tlc = TrafficLightColor.RED;
   }

   // Запуск имитации автоматизированного светофора
   public void run() {
      while(!stop) {
         try {
            switch(tlc) {
               case GREEN:
                  Thread.sleep(10000); // зеленый на 10 секунд
                  break;
               case YELLOW:
                  Thread.sleep(2000); // желтый на 2 секунды
                  break;
               case RED:
                  Thread.sleep(12000); // красный на 12 секунд
                  break;
            }
         } catch(InterruptedException exc) {
            System.out.println(exc);
         }
         changeColor();
      }
   }

   // Переключение цвета светофора
   synchronized void changeColor() {
      switch(tlc) {
         case RED:
            tlc = TrafficLightColor.GREEN;
            break;
         case YELLOW:
            tlc = TrafficLightColor.RED;
            break;
         case GREEN:
```

```
            tlc = TrafficLightColor.YELLOW;
      }

      changed = true;
      notify(); // уведомить о переключении цвета светофора
   }

   // Ожидание переключения цвета светофора
   synchronized void waitForChange() {
      try {
         while(!changed)
            wait(); // ожидать переключения цвета светофора
         changed = false;
      } catch(InterruptedException exc) {
         System.out.println(exc);
      }
   }

   // Возврат текущего цвета
   synchronized TrafficLightColor getColor() {
      return tlc;
   }

   // Прекращение имитации светофора
   synchronized void cancel() {
      stop = true;
   }
}

class TrafficLightDemo {
   public static void main(String args[]) {
      TrafficLightSimulator tl =
         new TrafficLightSimulator(TrafficLightColor.GREEN);
      Thread thrd = new Thread(tl);
      thrd.start();

      for(int i=0; i < 9; i++) {
         System.out.println(tl.getColor());
         tl.waitForChange();
      }

      tl.cancel();
   }
}
```

При выполнении этой программы на экран выводится показанный ниже результат. Как видите, цвета светофора переключаются в требуемой очередности: зеленый, желтый, красный.

```
GREEN
YELLOW
RED
```

```
GREEN
YELLOW
RED
GREEN
YELLOW
RED
```

Обратите внимание на то, что использование перечисления позволило упростить исходный код, нуждающийся в информации о состоянии светофора, и улучшить его структуризацию. Светофор может находиться в одном из трех состояний, и для этой цели в перечислении предусмотрены только три константы. Благодаря этому предотвращается случайное переключение имитируемого светофора в недопустимое состояние.

9. Можно усовершенствовать рассмотренную программу, используя тот факт, что перечисления реализуются в виде классов. Соответствующее задание будет предложено в упражнении для самопроверки в конце главы.

Автоупаковка

В версии JDK 5 были добавлены два очень полезных средства — *автоупаковка* и *автораспаковка*, — существенно упрощающие и ускоряющие создание кода, в котором приходится преобразовывать простые типы данных в объекты и наоборот. Поскольку такие ситуации возникают в программах на Java довольно часто, вытекающие отсюда преимущества почувствуют большинство программистов. Как будет продемонстрировано в главе 13, автоупаковка и автораспаковка в значительной мере обусловили широкую применимость обобщенных типов.

Автоупаковка и автораспаковка непосредственно связаны с оболочками типов и способами помещения значений в экземпляры оболочек и извлечения значений из них. Поэтому мы начнем с общего обзора оболочек типов и способов упаковки и распаковки значений вручную.

Оболочки типов

Как вы уже знаете, в Java предусмотрены простые типы данных, в том числе `int` и `double`. Простые типы позволяют добиться более высокой эффективности вычислений по сравнению с объектами. Однако простые типы не являются частью иерархии объектов и не наследуют свойства и методы класса `Object`.

Несмотря на высокую эффективность простых типов, возникают такие ситуации, когда для представления данных желательно использовать объекты. Например, переменную простого типа нельзя передать методу по ссылке. Кроме того, многие стандартные структуры данных, реализованные в Java, предполагают работу с объектами, и поэтому в них нельзя хранить данные простых типов. Для преодоления затруднений, возникающих в подобных и во многих других

ситуациях, в Java предусмотрены оболочки типов — классы, инкапсулирующие простые типы данных. Классы оболочек типов упоминались в главе 10, а здесь они будут рассмотрены более подробно.

Оболочки типов реализуются в классах `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character` и `Boolean`, входящих в пакет `java.lang`. Эти классы предоставляют методы, позволяющие полностью интегрировать простые типы данных в иерархию объектов Java.

Чаще всего применяются оболочки типов, представляющие числовые типы данных: `Byte`, `Short`, `Integer`, `Long`, `Float` и `Double`. Все оболочки числовых типов данных являются производными от абстрактного класса `Number`. В классе `Number` определены методы, возвращающие значение объекта для каждого числового типа данных.

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Например, метод `doubleValue()` возвращает значение объекта как `double`, метод `floatValue()` — как `float` и т.д. Перечисленные выше методы реализуются каждым классом оболочки числового типа.

В каждом классе оболочки числового типа предусмотрены конструкторы, позволяющие сформировать объект на основе соответствующего простого типа данных или его строкового представления. Например, в классах `Integer` и `Double` имеются следующие конструкторы.

```
Integer(int num)
Integer(String str)

Double(double num)
Double(String str)
```

Если параметр *str* не содержит допустимого строкового представления числового значения, то генерируется исключение `NumberFormatException`.

Во всех оболочках типов переопределен метод `toString()`, возвращающий из оболочки значение в удобной для чтения форме. Это позволяет выводить значения на экран, передавая объекты оболочек в качестве параметра методу, например `println()`, и не преобразуя их предварительно в простые типы данных.

Процесс инкапсуляции значения в оболочке типа называется *упаковкой*. До появления версии JDK 5 упаковка производилась вручную, т.е. посредством явного создания экземпляра класса оболочки с нужным значением. Например, для упаковки значения 100 в объект типа `Integer` требовалась следующая строка кода:

```
Integer iOb = new Integer(100);
```

В данном примере явно создается объект типа `Integer`, в который упаковывается значение 100, а ссылка на этот объект присваивается переменной `iOb`. В данном случае значение 100 упаковано в переменной `iOb`.

Процесс извлечения значения из объекта оболочки называется *распаковкой*. До появления версии JDK 5 распаковка также выполнялась вручную, т.е. для извлечения значения, упакованного в этом объекте, приходилось явным образом вызывать соответствующий метод объекта оболочки. Например, для распаковки значения из объекта `iOb` вручную и присваивания результата переменной `int` требовалась следующая строка кода:

```
int i = iOb.intValue();
```

В данном примере метод `intValue()` возвращает значение, упакованное в объекте `iOb` как `int`.

Рассмотренные выше механизмы упаковки и распаковки демонстрируются в приведенном ниже примере программы.

```
// Упаковка и распаковка значений вручную
class Wrap {
   public static void main(String args[]) {

      Integer iOb = new Integer(100) ;  ◄——— Ручная упаковка значения 100

      int i = iOb.intValue() ;  ◄——— Ручная распаковка значения в iOb

      System.out.println(i + " " + iOb); // отображает 100 100
   }
}
```

В данной программе целочисленное значение 100 упаковывается в объект типа `Integer`, на который ссылается переменная `iOb`. Для извлечения упакованного числового значения вызывается метод `intValue()`. Полученное значение сохраняется в переменной `i`. А в конце программы на экран выводятся значения переменных `i` и `iOb`, каждое из которых равно 100.

Аналогичная процедура использовалась в программах для упаковки и распаковки значений, начиная с ранних версий Java и до появления JDK 5. Но это не совсем удобно. Более того, создание объектов оболочек разных типов вручную может сопровождаться ошибками. Но теперь, с появлением автоупаковки и автораспаковки, обращаться с оболочками типов стало значительно проще.

Основные сведения об автоупаковке

Автоупаковка — это процесс автоматической инкапсуляции (упаковки) простого типа данных в объектную оболочку соответствующего типа всякий раз, когда в этом возникает необходимость, причем создавать такой объект явным образом не нужно. *Автораспаковка* — это обратный процесс автоматического извлечения (распаковки) значения, упакованного в объектную оболочку. Благодаря автораспаковке отпадает необходимость в вызове таких методов, как `intValue()` и `doubleValue()`.

Поддержка автоупаковки и autoраспаковки существенно упрощает реализацию целого ряда алгоритмов, так как в этом случае все рутинные операции по упаковке и распаковке значений простых типов берет на себя исполняющая среда Java, что позволяет уменьшить вероятность возникновения программных ошибок. Автоупаковка освобождает программиста от необходимости создавать вручную объекты для заключения в них простых типов данных. Достаточно присвоить упаковываемое значение переменной, ссылающейся на объект оболочки соответствующего типа, и нужный объект будет автоматически создан исполняющей средой Java. Ниже приведен пример создания объекта типа `Integer`, в который автоматически упаковывается целочисленное значение 100.

```
Integer iOb = 100; // автоупаковка целочисленного значения
```

Обратите внимание на то, что в данном примере отсутствует оператор `new`, конструирующий объект явным образом. Создание объекта происходит автоматически.

Для распаковки значения из объекта достаточно присвоить переменной простого типа ссылку на этот объект. Например, для распаковки значения, упакованного в объекте `iOb`, нужно ввести в код следующую строку:

```
int i = iOb; // автораспаковка
```

Все остальное возьмет на себя исполняющая среда Java. Ниже приведен пример программы, демонстрирующий автоупаковку и автораспаковку.

```
// Демонстрация автоупаковки и автораспаковки
class AutoBox {
   public static void main(String args[]) {

      Integer iOb = 100; // автоупаковка целого числа   ◄─┐ Автоупаковка и
                                                          ├─ автораспаковка
      int i = iOb; // автораспаковка ◄────────────────────┘ значения 100

      System.out.println(i + " " + iOb); // отображает 100 100
   }
}
```

Автоупаковка и методы

Автоупаковка и автораспаковка происходят не только в простых операциях присваивания, но и в тех случаях, когда простой тип требуется преобразовать в объект и наоборот. Следовательно, автоупаковка и автораспаковка могут происходить при передаче аргумента методу и при возврате значения последним. Рассмотрим в качестве примера следующую программу.

```
// Автоупаковка и автораспаковка при передаче
// параметров и возврате значений из методов.

class AutoBox2 {
   // Этот метод имеет параметр типа Integer
   static void m(Integer v) { ◄──────── Получение параметра Integer
```

```
      System.out.println("m() получил " + v);
   }

   // Этот метод возвращает значение типа int
   static int m2() { ◄———————————— Возврат значения типа int
      return 10;
   }

   // Этот метод возвращает значение типа Integer
   static Integer m3() { ◄———————————— Возврат значения типа Integer
      return 99; // автоупаковка значения 99 в объект типа Integer
   }

   public static void main(String args[]) {

      // Передача методу m() значения типа int.
      // Метод m() имеет параметр типа Integer,
      // поэтому значение int автоматически упаковывается.
      m(199);

      // Объект iOb получает значение типа int, возвращаемое
      // методом m2(). Это значение автоматически упаковывается,
      // чтобы его можно было присвоить объекту iOb.
      Integer iOb = m2();
      System.out.println("Значение, возвращенное из m2(): " + iOb);

      // Далее метод m3() возвращает значение типа Integer, которое
      // автоматически распаковывается и преобразуется в тип int.
      int i = m3();
      System.out.println("Значение, возвращенное из m3(): " + i);

      // Далее методу Math.sqrt() в качестве параметра передается
      // объект iOb, который автоматически распаковывается, а его
      // значение повышается до типа double, требующегося для
      // выполнения данного метода.
      iOb = 100;
      System.out.println("Корень квадратный из iOb: " +
                         Math.sqrt(iOb));
   }
}
```

Результат выполнения данной программы выглядит так.

```
m() получил 199
Значение, возвращенное из m2(): 10
Значение, возвращенное из m3(): 99
Корень квадратный из iOb: 10.0
```

В объявлении метода `m()` указывается, что ему должен передаваться параметр типа `Integer`. В методе `main()` целочисленное значение 199 передается методу `m()` в качестве параметра. В итоге происходит автоупаковка этого целочисленного значения. Далее в программе вызывается метод `m2()`,

возвращающий целочисленное значение 10, которое присваивается переменной ссылки на объект iOb в методе main(). А поскольку объект iOb относится к типу Integer, то целочисленное значение, возвращаемое методом m2(), автоматически упаковывается. Затем в методе main() вызывается метод m3(). Он возвращает объект типа Integer, который посредством autoраспаковки преобразуется в тип int. И наконец, в методе main() вызывается метод Math.sqrt(), которому в качестве аргумента передается объект iOb. В данном случае происходит автораспаковка данного объекта, а его значение повышается до типа double, поскольку параметр именно этого типа должен быть передан методу Math.sqrt().

Автоупаковка и автораспаковка в выражениях

Автоупаковка и автораспаковка выполняются всякий раз, когда требуется преобразовать простой тип в объект, а объект — в простой тип. Так, автораспаковка происходит при вычислении выражений, и если это требуется, то результат вычисления упаковывается. Рассмотрим в качестве примера приведенную ниже программу.

```
// Автоупаковка и автораспаковка в выражениях

class AutoBox3 {
   public static void main(String args[]) {
      Integer iOb, iOb2;
      int i;

      iOb = 99;
      System.out.println("Исходное значение iOb: " + iOb);

      // В следующем выражении объект iOb автоматически
      // распаковывается, производятся вычисления, а результат
      // снова упаковывается в объект iOb
      ++iOb;
      System.out.println("После ++iOb: " + iOb);

      // Здесь выполняется автораспаковка объекта iOb,
      // к полученному значению прибавляется число 10,
      // а результат снова упаковывается в объект iOb
      iOb += 10;
      System.out.println("После iOb += 10: " + iOb);

      // Выполняется автораспаковка объекта iOb, производятся
      // вычисления, а результат снова упаковывается в объект iOb
      iOb2 = iOb + (iOb / 3);
      System.out.println("iOb2 после вычисления выражения: " + iOb2);

      // Вычисляется то же самое выражение,
      // но повторная упаковка не выполняется
      i = iOb + (iOb / 3);
```

Примеры автоупаковки/автораспаковки в выражениях

```
        System.out.println("i после вычисления выражения: " + i);
    }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Исходное значение iOb: 99
После ++iOb: 100
После iOb += 10: 110
iOb2 после вычисления выражения: 146
i после вычисления выражения: 146
```

Обратите внимание на следующую строку кода программы:

```
++iOb;
```

В ней значение объекта iOb должно быть увеличено на единицу. Происходит это следующим образом: объект iOb распаковывается, полученное значение инкрементируется, а результат снова упаковывается в объект iOb.

Благодаря автораспаковке объектные оболочки целочисленных типов, например Integer, можно использовать в инструкции switch. В качестве примера рассмотрим следующий фрагмент кода.

```
Integer iOb = 2;

switch(iOb) {
   case 1: System.out.println("один");
      break;
   case 2: System.out.println("два");
      break;
   default: System.out.println("ошибка");
}
```

При вычислении выражения в инструкции switch объект iOb распаковывается, и последующей обработке подвергается значение типа int, упакованное в этом объекте.

Как следует из приведенных выше примеров, выражения, в которых применяются объектные оболочки простых типов, становятся интуитивно понятными благодаря автоупаковке и автораспаковке. До появления версии JDK 5 для достижения аналогичного результата в программе приходилось прибегать к приведению типов и вызовам специальных методов вроде intValue().

Предупреждение относительно использования автоупаковки и автораспаковки

Поскольку автоупаковка и автораспаковка предельно упрощают обращение с оболочками простых типов, может возникнуть соблазн всегда использовать вместо простых типов только их оболочки, например Integer или Double. Так, например, автоупаковка и автораспаковка позволяют создавать код, подобный следующему.

```
// Неоправданное использование автоупаковки и автораспаковки
Double a, b, c;

a = 10.2;
b = 11.4;
c = 9.8;

Double avg = (a + b + c) / 3;
```

В данном примере в объектах типа Double хранятся три значения, используемые для вычисления арифметического среднего, а полученный результат присваивается другому объекту типа Double. И хотя такой код формально считается корректным, а следовательно, будет выполняться правильно, применение в нем автоупаковки и автораспаковки ничем не оправданно. Ведь подобный код значительно менее эффективен, чем аналогичный код, написанный только с использованием переменных типа double. С любой из операций распаковки и упаковки связаны издержки, отсутствующие при использовании простых типов.

В целом оболочки типов следует использовать только тогда, когда объектное представление простых типов действительно необходимо. Ведь автоупаковка и автораспаковка добавлены в Java не для того, чтобы сделать ненужными простые типы.

Статический импорт

Java поддерживает расширенное использование ключевого слова import. Указав после слова import ключевое слово static, можно импортировать статические члены класса или интерфейса. Данная возможность обеспечивается механизмом *статического импорта*. Статический импорт позволяет ссылаться на статические члены по их простым именам, без дополнительного указания имен классов, что упрощает синтаксис.

Для того чтобы оценить по достоинству возможности статического импорта, начнем с примера, в котором это средство не используется. Ниже проведен пример программы для решения следующего квадратного уравнения:

$$ax^2 + bx + c = 0$$

В этой программе используются два статических метода — Math.pow() и Math.sqrt() — из класса Math, который, в свою очередь, входит в пакет java.lang. Первый из методов возвращает значение, возведенное в заданную степень, а второй — квадратный корень переданного значения.

```
// Нахождение корней квадратного уравнения
class Quadratic {
   public static void main(String args[]) {

      // a, b и c представляют коэффициенты
      // квадратного уравнения ax^2 + bx + c = 0
      double a, b, c, x;
```

```
      // Решить квадратное уравнение 4x² + x - 3 = 0
      a = 4;
      b = 1;
      c = -3;

      // Найти первый корень
      x = (-b + Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
      System.out.println("Первый корень: " + x);

      // Найти второй корень
      x = (-b - Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
      System.out.println("Второй корень: " + x);
   }
}
```

Методы pow() и sqrt() — статические, а следовательно, их нужно вызывать, ссылаясь на имя класса Math. Их вызов осуществляется в следующем выражении, которое выглядит довольно громоздким:

```
x = (-b + Math.sqrt(Math.pow(b, 2) - 4 * a * c)) / (2 * a);
```

В выражениях подобного типа приходится постоянно следить за тем, чтобы перед методами pow() и sqrt() (и другими подобными методами, например sin(), cos() и tan()) было указано имя класса, что неудобно и чревато ошибками.

Утомительной обязанности указывать всякий раз имя класса перед статическим методом позволяет избежать статический импорт. Его применение демонстрирует приведенная ниже переработанная версия предыдущей программы.

```
// Использование статического импорта для
// помещения методов sqrt() и pow() в область видимости
import static java.lang.Math.sqrt;   <-- Использование статического импорта
import static java.lang.Math.pow;    <-- для помещения методов sqrt()
                                         и pow() в область видимости

class Quadratic {
   public static void main(String args[]) {

      // a, b и c представляют коэффициенты квадратного уравнения
      // ax2 + bx + c = 0
      double a, b, c, x;

      // Решить квадратное уравнение 4x2 + x - 3 = 0
      a = 4;
      b = 1;
      c = -3;

      // Найти первый корень
      x = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
      System.out.println("Первый корень: " + x);

      // Найти второй корень
      x = (-b - sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
      System.out.println("Второй корень: " + x);
   }
}
```

В данной версии программы имена методов sqrt и pow уже не нужно указывать полностью (т.е. вместе с именем их класса). И достигается это благодаря статическому импорту обоих методов в приведенных ниже инструкциях, делающих оба метода непосредственно доступными.

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

Добавление этих строк избавляет от необходимости предварять имена методов sqrt() и pow() именем их класса. В итоге выражение для решения квадратного уравнения принимает следующий вид:

```
x = (-b + sqrt(pow(b, 2) - 4 * a * c)) / (2 * a);
```

Теперь оно выглядит проще и воспринимается легче.

В Java предусмотрены две общие формы инструкции import static. В первой форме, использованной в предыдущем примере, непосредственно доступным для программы делается единственное имя. Ниже приведена эта общая форма статического импорта:

```
import static пакет.имя_типа.имя_статического_члена;
```

где *имя_типа* обозначает класс или интерфейс, содержащий требуемый статический член, на который указывает *имя_статического_члена*. Вторая общая форма инструкции статического импорта выглядит так:

```
import static пакет.имя_типа.*;
```

Если предполагается использовать несколько статических методов или полей, определенных в классе, то данная общая форма записи позволяет импортировать все эти члены одновременно. Таким образом, обеспечить непосредственный доступ к методам pow() и sqrt() в предыдущей версии программы (а также к другим статическим членам класса Math) без указания имени класса можно с помощью следующей единственной строки кода:

```
import static java.lang.Math.*;
```

Очевидно, что статический импорт не ограничивается только классом Math и его методами. Так, если требуется сделать непосредственно доступным статическое поле System.out потока стандартного вывода, достаточно ввести в программу следующую строку кода:

```
import static java.lang.System.out;
```

После этого данные можно выводить на консоль, не указывая перед статическим полем out имя его класса System.

```
out.println("Импортировав System.out, имя out можно использовать
            непосредственно.");
```

Насколько целесообразно поступать именно так — вопрос спорный. С одной стороны, размер исходного кода в этом случае сокращается. А с другой стороны, тем, кто просматривает исходный код программы, может быть непонятно, что конкретно обозначает имя out: поток стандартного вывода System.out или нечто иное.

Каким бы удобным ни был статический импорт, важно следить за тем, чтобы он применялся корректно. Как известно, библиотеки классов Java организованы в пакеты именно для того, чтобы исключить конфликты имен. Если импортируются статические члены класса, то они переносятся в глобальное пространство имен. Вследствие этого увеличивается вероятность конфликтов и непреднамеренного сокрытия имен. Если статический член используется в программе один или два раза, то импортировать его нет никакого смысла. Кроме того, некоторые имена статических членов (например, `System.out`) настолько знакомы всем программирующим на Java, что они окажутся менее узнаваемыми, если будут использоваться без имени своего класса. Статический импорт был добавлен в Java в расчете на программы, в которых постоянно используются определенные статические члены. Следовательно, к статическому импорту следует прибегать осмотрительно, не злоупотребляя им.

Спросим у эксперта

ВОПРОС. Допускается ли статический импорт статических членов самостоятельно создаваемых классов?

ОТВЕТ. Допускается. Это средство можно применять для импорта статических членов любых создаваемых вами классов и интерфейсов. Им особенно удобно пользоваться в тех случаях, когда в классе определено несколько статических членов, часто используемых в большой программе. Так, если в классе определен ряд констант, обозначающих граничные значения, то статический импорт позволит избежать излишнего упоминания их классов.

Аннотации (метаданные)

Java предоставляет возможность внедрять в исходный файл дополнительную информацию в виде *аннотаций*, не изменяя поведения программы. Эта информация может быть использована различными инструментальными средствами как на этапе разработки, так и в процессе развертывания программы. В частности, аннотации могут обрабатываться генератором исходного кода, компилятором и средствами развертывания приложений. Дополнительные сведения, включаемые в исходный файл, также называют *метаданными*, но термин "аннотация" представляется более описательным и чаще используется.

Полное рассмотрение аннотаций выходит за рамки данной книги. Для подробного их рассмотрения здесь просто недостаточно места. Поэтому ограничимся лишь кратким описанием самого понятия и назначения аннотаций.

Примечание

Подробнее о метаданных и аннотациях можно прочитать в книге *Java. Полное руководство, 10-е издание*.

Аннотации создаются посредством механизма, основанного на интерфейсах. Ниже приведен простой пример аннотации.

```
// Простой пример аннотации
@interface MyAnno {
   String str();
   int val();
}
```

В данном примере объявляется аннотация MyAnno. Заметьте, что ключевое слово interface предваряется знаком @. Тем самым компилятору сообщается об объявлении аннотации. Обратите внимание на два члена: str() и val(). Все аннотации содержат лишь объявления методов без определения их тел. Объявленные методы реализует исполняющая среда Java, причем они действуют во многом подобно полям.

Аннотации всех типов автоматически расширяют интерфейс Annotation. Следовательно, интерфейс Annotation используется в качестве суперинтерфейса для всех аннотаций. Он входит в пакет java.lang.annotation.

Первоначально аннотации использовались только для аннотирования объявлений. При этом аннотацию можно связать с любым объявлением. В частности, аннотированными могут быть объявления классов, методов, полей, параметров, констант перечислимого типа и даже самих аннотаций. Но в любом случае аннотация предшествует остальной части объявления. Начиная с версии JDK 8 можно аннотировать также *использование типов*, например использование приводимого или возвращаемого методом типа.

Вводя аннотацию, вы задаете значения ее членов. Ниже приведен пример применения аннотации MyAnno к методу.

```
// Аннотирование метода
@MyAnno(str = "Пример аннотации", val = 100)
public static void myMeth() { // ...
```

Данная аннотация связывается с методом myMeth(). Изучите ее синтаксис. Имени аннотации предшествует знак @, а после имени следует заключенный в скобки список инициализируемых членов. Для того чтобы задать значение члена аннотации, следует присвоить это значение имени данного члена. В рассматриваемом здесь примере строка "Пример аннотации" присваивается члену str аннотации MyAnno. Обратите внимание на отсутствие скобок после идентификатора str в этом присваивании. При присваивании значения члену аннотации используется только его имя. В этом отношении члены аннотации похожи на поля.

Аннотация без параметров называется *маркерной аннотацией*. При определении маркерной аннотации круглые скобки не указываются. Главное ее назначение — пометить объявление некоторым атрибутом.

В Java определено множество встроенных аннотаций. Большинство из них являются специализированными, но есть девять аннотаций общего назначения. Четыре из них входят в пакет `java.lang.annotation` — это аннотации `@Retention`, `@Documented`, `@Target` и `@Inherited`. Пять аннотаций — `@Override`, `@Deprecated`, `@SafeVarargs`, `@FunctionalInterface` и `@SuppressWarnings` — включены в пакет `java.lang`. Все эти аннотации приведены в табл. 12.1.

Таблица 12.1. Встроенные аннотации

Аннотация	Описание
`@Retention`	Задает стратегию управления жизненным циклом аннотации. Эта стратегия определяет, будет ли видна аннотация в исходном коде, скомпилированном файле и в процессе выполнения
`@Documented`	Маркерная аннотация, сообщающая инструментальному средству о том, что аннотация должна документироваться. Эту аннотацию следует использовать только для аннотирования объявления другой аннотации
`@Target`	Задает виды объявлений, к которым может применяться аннотация. Данная аннотация предназначена только для использования в отношении другой аннотации. Она получает аргумент в виде константы или массива констант перечислимого типа `ElementType`, таких как `CONSTRUCTOR`, `FIELD` и `METHOD`. Аргумент определяет виды объявлений, к которым может быть применена аннотация. Отсутствие аннотации `@Target` указывает на то, что данная аннотация может применяться к любому объявлению
`@Inherited`	Маркерная аннотация, указывающая на то, что аннотация суперкласса должна наследоваться подклассом
`@Override`	Метод, аннотированный как `@Override`, должен переопределять метод суперкласса. Если это условие не выполняется, то возникает ошибка компиляции. Данная аннотация представляет собой маркер и позволяет убедиться в том, что метод суперкласса действительно переопределен, а не перегружен
`@Deprecated`	Маркерная аннотация, указывающая на то, что объявление устарело и было заменено новым
`@SafeVarargs`	Маркерная аннотация, которая указывает на то, что в методе или конструкторе не выполняются действия, небезопасные с точки зрения использования переменного количества аргументов. Может применяться только к статическим или финальным методам и конструкторам

Окончание табл. 12.1

Аннотация	Описание
`@FunctionalInterface`	Маркерная аннотация, используемая для аннотирования объявлений интерфейсов. Она указывает на то, что аннотируемый ею интерфейс является ***функциональным интерфейсом***, который содержит один и только один абстрактный метод. Функциональные интерфейсы используются в лямбда-выражениях. (Функциональные интерфейсы будут подробно рассмотрены в главе 14.) Важно понимать, что аннотация `@FunctionalInterface` играет исключительно информационную роль. Любой интерфейс, имеющий ровно один абстрактный метод, по определению является функциональным интерфейсом
`@SuppressWarnings`	Указывает на то, что одно или более предупреждающих сообщений, которые могут быть сгенерированы в процессе компиляции, должны подавляться. Подавляемые предупреждающие сообщения задаются именами, представляемыми в виде строк

Примечание

Кроме аннотаций, входящих в пакет `java.lang.annotation`, в JDK 8 определены аннотации `@Repeatable` и `@Native`. Аннотация `@Repeatable` обеспечивает поддержку повторяющихся аннотаций, для которых возможно более чем однократное применение к одному и тому же элементу. Аннотация `@Native` используется для аннотирования постоянного поля, к которому имеет доступ исполняемый (т.е. собственный машинный) код. Обе аннотации имеют специфическую сферу применения, и их рассмотрение выходит за рамки книги.

Ниже приведен пример, в котором аннотацией `@Deprecated` помечены класс `MyClass` и метод `getMsg()`. При попытке скомпилировать программу будет выведено сообщение о том, что в исходном коде содержатся устаревшие и не рекомендованные к применению элементы.

```
// Пример использования аннотации @Deprecated

// Пометить класс как не рекомендованный к применению
@Deprecated ◄——— Пометить класс как не рекомендованный к применению
class MyClass {
   private String msg;

   MyClass(String m) {
      msg = m;
   }

   // Пометить метод как не рекомендованный к применению
   @Deprecated ◄——— Пометить метод как не рекомендованный к применению
```

```
   String getMsg() {
      return msg;
   }

   // ...
}

class AnnoDemo {
   public static void main(String args[]) {
      MyClass myObj = new MyClass("тест");

      System.out.println(myObj.getMsg());
   }
}
```

Вопросы и упражнения для самопроверки

1. Константы перечислимого типа иногда называют *самотипизированными*. Что это означает?
2. Какой класс автоматически наследуют перечисления?
3. Напишите для приведенного ниже перечисления программу, в которой метод `values()` используется для отображения списка констант и их значений.

```
enum Tools {
   SCREWDRIVER, WRENCH, HAMMER, PLIERS
}
```

4. Созданную в упражнении 12.1 программу, имитирующую автоматизированный светофор, можно усовершенствовать, внеся ряд простых изменений, позволяющих выгодно воспользоваться возможностями перечислений. В исходной версии этой программы продолжительность отображения каждого цвета светофора регулировалась в классе `TrafficLightSimulator`, причем значения задержек были жестко запрограммированы в методе `run()`. Измените исходный код программы таким образом, чтобы продолжительность отображения каждого цвета светофора задавалась константами перечислимого типа `TrafficLightColor`. Для этого вам понадобятся конструктор, переменная экземпляра, объявленная как `private`, и метод `getDelay()`. Подумайте о том, как еще можно улучшить данную программу. (Подсказка: попробуйте отказаться от инструкции `switch` и воспользоваться порядковыми значениями каждого цвета для переключения светофора.)
5. Что такое упаковка и распаковка? В каких случаях выполняется автоупаковка и автораспаковка?

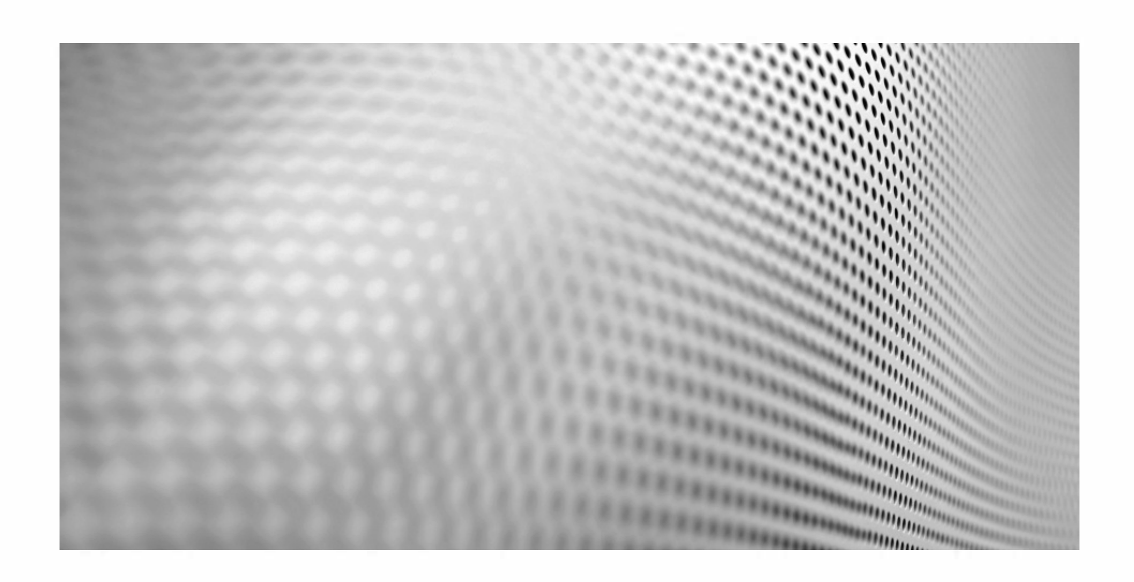
6. Измените следующий фрагмент кода таким образом, чтобы в нем выполнялась автоупаковка:

```
Double val = Double.valueOf(123.0);
```

7. Объясните, что такое статический импорт.
8. Какие действия выполняет приведенная ниже инструкция?

```
import static java.lang.Integer.parseInt;
```

9. Следует ли использовать статический импорт применительно к конкретным ситуациям или желательно импортировать статические члены всех классов?
10. Синтаксис аннотации основывается на ________________.
11. Какая аннотация называется маркерной?
12. Справедливо ли следующее утверждение: “Аннотации применимы только к методам”?



Глава 13

Обобщения

В этой главе...

- Преимущества обобщений
- Создание обобщенного класса
- Ограниченные параметры типов
- Шаблоны аргументов
- Применение ограниченных шаблонов
- Создание обобщенного метода
- Создание обобщенного конструктора
- Создание обобщенного интерфейса
- Использование базовых типов
- Выведение типов
- Очистка
- Исключение ошибок неоднозначности
- Ограничения обобщений

После выхода первоначальной версии 1.0 в язык Java было добавлено множество новых средств. Каждое нововведение расширяло возможности языка и сферу его применения, однако одно из них имело особенно далеко идущие последствия. Речь идет об *обобщениях* — абсолютно новой синтаксической конструкции, введение которой повлекло за собой существенные изменения во многих классах и методах ядра API. Не будет преувеличением сказать, что обобщения коренным образом изменили сам язык Java.

Обобщения — слишком обширная тема, чтобы ее можно было полностью рассмотреть в рамках данной книги, однако понимание базовых возможностей этого средства необходимо каждому, кто программирует на Java. Поначалу синтаксис обобщений может показаться вам удручающе сложным, но пусть вас это не смущает. В действительности обобщения на удивление просты в использовании. К тому моменту, когда вы завершите чтение данной главы, вы не только усвоите все ключевые понятия, но и научитесь успешно применять обобщения в своих программах.

Основные сведения об обобщениях

Термин *обобщение* по сути означает *параметризированный тип*. Специфика параметризированных типов состоит в том, что они позволяют создавать

классы, интерфейсы и методы, в которых тип данных указывается в виде параметра. Используя обобщения, можно создать единственный класс, который будет автоматически работать с различными типами данных. Классы, интерфейсы и методы, оперирующие параметризированными типами, называются *обобщенными*, как, например, *обобщенный класс* или *обобщенный метод*.

Главное преимущество обобщенного кода состоит в том, что он будет автоматически работать с типом данных, переданным ему в качестве параметра. Многие алгоритмы выполняются одинаково, независимо от того, к данным какого типа они будут применяться. Например, быстрая сортировка не зависит от типа данных, будь то `Integer`, `String`, `Object` или `Thread`. Используя обобщения, можно реализовать алгоритм один раз, а затем применять его без дополнительных усилий к любому типу данных.

Следует особо подчеркнуть, что в Java всегда была возможность создавать обобщенный код, оперирующий ссылками типа `Object`. А поскольку класс `Object` выступает в качестве суперкласса по отношению ко всем остальным классам, то ссылки на тип `Object` позволяют обращаться к объекту любого типа. Таким образом, еще до появления обобщений можно было оперировать разнотипными объектами посредством одной переменной с помощью ссылок на тип `Object`. Проблема состояла в том, что такой подход, требующий явного преобразования типа `Object` в конкретный тип посредством приведений, не обеспечивал безопасность типов. Это служило потенциальным источником ошибок из-за того, что приведение типов могло быть неумышленно выполнено неверно. Обобщения обеспечивают безопасность типов, которой раньше так недоставало, поскольку в этом случае автоматически выполняются неявные приведения. Таким образом, обобщения расширяют возможности повторного использования кода, делая этот процесс безопасным и надежным.

Спросим у эксперта

ВОПРОС. Говорят, что обобщения в Java аналогичны шаблонам в С++. Так ли это?

ОТВЕТ. Действительно, обобщения в Java похожи на шаблоны в С++. То, что в Java называют параметризованными типами, в С++ называют шаблонами. Однако эти понятия не являются эквивалентными. Между ними имеется ряд принципиальных различий, а в целом обобщения в Java намного более просты в применении.

Тем, у кого имеется опыт программирования на С++, важно помнить, что навыки применения шаблонов нельзя механически переносить на обобщения в Java. У этих языковых средств имеются отличия, которые слабо проявляются внешне, но довольно значительны по сути.

Простой пример обобщений

Прежде чем приступить к более подробному изучению обобщений, полезно рассмотреть простой пример их применения. Ниже приведен исходный код программы, в которой объявлены два класса: обобщенный класс Gen и использующий его класс GenDemo.

```
// Простой обобщенный класс.
// Здесь T - это параметр типа, вместо которого
// при создании объекта класса Gen будет подставляться
// реально существующий тип.

class Gen<T> {  ◄── В объявлении этого класса T означает обобщенный тип
   T ob; // объявить объект типа T

   // Передать конструктору ссылку на объект типа T
   Gen(T o) {
      ob = o;
   }

   // Вернуть объект ob из метода
   T getob() {
      return ob;
   }

   // Отобразить тип T
   void showType() {
      System.out.println("Тип T - это " + ob.getClass().getName());
   }
}

// Демонстрация использования обобщенного класса
class GenDemo {
   public static void main(String args[]) {
      // Создать обобщенную ссылку на целочисленное значение
      Gen<Integer> iOb;  ◄── Создание ссылки на объект типа Gen<Integer>

      // Создать объект типа Gen<Integer> и присвоить ссылку
      // на него переменной iOb. Обратите внимание на
      // автоупаковку при инкапсуляции значения 88 в объекте
      // типа Integer.
      iOb = new Gen<Integer>(88);  ◄── Создание экземпляра типа Gen<Integer>

      // Отобразить тип данных, используемых в объекте iOb
      iOb.showType();

      // Получить значения из объекта iOb. Обратите внимание
      // на то, что приведение типов здесь не требуется.
      int v = iOb.getob();
      System.out.println("значение: " + v);
```

```
        System.out.println();
                                                          Создание ссылки и объекта
                                                                  типа Gen<String>
        // Создать объект типа Gen для строк
        Gen<String> strOb = new Gen<String>("Тестирование обобщений");

        // Отобразить тип данных, используемых в объекте strOb
        strOb.showType();

        // Получить значение из объекта strOb. Заметьте,
        // что приведение типов здесь также не требуется.
        String str = strOb.getob();
        System.out.println("значение: " + str);
    }
}
```

В результате выполнения данной программы будет получен следующий результат.

```
Тип T - это java.lang.Integer
значение: 88

Тип T - это java.lang.String
значение: Тестирование обобщений
```

Рассмотрим исходный код программы более подробно. Прежде всего обратите внимание на способ объявления класса Gen. Для этого используется следующая строка кода:

```
class Gen<T> {
```

где T — имя параметра типа. Это имя — заполнитель, подлежащий замене фактическим типом, передаваемым конструктору Gen() при создании объекта. Следовательно, имя T применяется в классе Gen всякий раз, когда возникает необходимость в использовании параметра типа. Обратите внимание на то, что имя T заключено в угловые скобки (<>). Этот синтаксис является общим: всякий раз, когда объявляется параметр типа, он указывается в угловых скобках. Поскольку класс Gen использует параметр типа, он является обобщенным классом.

В объявлении класса Gen имя для параметра типа могло быть выбрано совершенно произвольно, но по традиции выбирается имя T. Вообще говоря, для этого рекомендуется выбирать имя, состоящее из одной прописной буквы. Другими распространенными именами параметров типа являются V и E.

Далее в программе имя T используется при объявлении объекта ob:

```
T ob; // объявить объект типа T
```

Как уже отмечалось, имя параметра типа T служит заполнителем, вместо которого при создании объекта класса Gen указывается конкретный тип. Поэтому объект ob будет иметь тип, передаваемый в виде параметра T при получении экземпляра объекта класса Gen. Так, если в качестве параметра типа T указывается String, то объект ob будет иметь тип String.

Рассмотрим конструктор класса Gen.

```
Gen(T o) {
   ob = o;
}
```

Отсюда следует, что параметр o конструктора имеет тип T. Это означает, что конкретный тип параметра o определяется типом, передаваемым в виде параметра T при создании объекта класса Gen. А поскольку параметр o и переменная экземпляра ob относятся к типу T, то после создания объекта класса Gen их конкретный тип окажется одним и тем же.

Кроме того, параметр типа T можно указывать в качестве типа значения, возвращаемого методом.

```
T getob() {
   return ob;
}
```

Переменная экземпляра ob также относится к типу T, поэтому ее тип совпадает с типом значения, возвращаемого методом getob().

Метод showType() отображает тип T. Это делается путем вызова метода getName() для объекта типа Class, возвращаемого вызовом метода getClass() для объекта ob. Поскольку до этого мы еще ни разу не использовали такую возможность, рассмотрим ее более подробно. Как объяснялось в главе 7, в классе Object определен метод getClass(). Следовательно, этот метод является членом класса любого типа. Он возвращает объект типа Class, соответствующий классу объекта, для которого он вызван. Класс Class, определенный в пакете java.lang, инкапсулирует информацию о текущем классе. Он имеет несколько методов, которые позволяют получать информацию о классах во время выполнения. К их числу принадлежит метод getName(), возвращающий строковое представление имени класса.

В классе GenDemo демонстрируется использование обобщенного класса Gen. Прежде всего, в нем создается версия класса Gen для целых чисел:

```
Gen<Integer> iOb;
```

Внимательно проанализируем это объявление. Заметьте, что в угловых скобках после имени класса Gen указан тип Integer. В данном случае Integer — это аргумент типа, передаваемый параметру T класса Gen. В конечном счете мы получаем класс Gen, в котором везде, где был указан тип T, теперь фигурирует тип Integer. Следовательно, после такого объявления типом переменной ob и возвращаемым типом метода getob() становится тип Integer.

Прежде чем мы начнем продвигаться дальше, важно подчеркнуть, что на самом деле никакие разные версии класса Gen (как и вообще любого другого класса) компилятором Java не создаются. В действительности компилятор просто удаляет всю информацию об обобщенном типе, выполняя все необходимые приведения типов и тем самым заставляя код вести себя так, словно была создана специфическая версия класса Gen, хотя в действительности в программе

существует только одна версия Gen — обобщенная. Процесс удаления информации об обобщенном типе называется *очисткой*, и к этой теме мы еще вернемся в данной главе.

В следующей строке кода переменной iOb присваивается ссылка на экземпляр, соответствующий версии класса Gen для типа Integer:

```
iOb = new Gen<Integer>(88);
```

Обратите внимание на то, что при вызове конструктора класса Gen указывается также аргумент типа Integer. Это необходимо потому, что тип объекта, на который указывает ссылка (в данном случае — iOb), должен соответствовать Gen<Integer>. Если тип ссылки, возвращаемой оператором new, будет отличаться от Gen<Integer>, то компилятор сообщит об ошибке. Например, это произойдет при попытке скомпилировать следующую строку кода:

```
iOb = new Gen<Double>(88.0); // Ошибка!
```

Переменная iOb относится к типу Gen<Integer>, а следовательно, она не может быть использована для хранения ссылки на объект типа Gen<Double>. Этот вид проверки — одно из основных преимуществ обобщенных типов, поскольку они обеспечивают безопасность типов.

Как следует из комментариев к программе, в рассматриваемом здесь операторе присваивания осуществляется автоупаковка целочисленного значения 88 в объект типа Integer:

```
iOb = new Gen<Integer>(88);
```

Это происходит потому, что обобщение Gen<Integer> создает конструктор, которому передается аргумент типа Integer. А поскольку предполагается создание объекта типа Integer, то в нем автоматически упаковывается целочисленное значение 88. Разумеется, все это можно было бы явно указать в операторе присваивания, как показано ниже.

```
iOb = new Gen<Integer>(new Integer(88));
```

Однако получаемая в этом случае длинная строка кода не дает никаких преимуществ по сравнению с предыдущей, более компактной записью.

Затем программа отображает тип переменной ob, инкапсулированной в объекте iOb (в данном случае это тип Integer). Значение переменной ob получается в следующей строке кода:

```
int v = iOb.getob();
```

Метод getob() возвращает значение типа T, замененное на Integer при объявлении переменной, ссылающейся на объект iOb, а следовательно, метод getob() фактически возвращает значение того же самого типа Integer. Это значение автоматически распаковывается, прежде чем оно будет присвоено переменной v типа int.

И наконец, в классе GenDemo объявляется объект типа Gen<String>.

```
Gen<String> strOb = new Gen<String>("Generics Test");
```

Поскольку аргументом типа является String, этот класс заменит параметр T во всем коде класса Gen. В результате создается (концептуально) строковая версия класса Gen, что и демонстрируют остальные строки кода программы.

Обобщения работают только с объектами

Когда объявляется экземпляр обобщенного типа, аргумент, передаваемый параметру типа, должен быть типом класса. Использовать для этой цели простые типы, например int или char, нельзя. Например, классу Gen можно передать через параметр T любой тип класса, но передача любого простого типа недопустима. Иными словами, следующее объявление приведет к ошибке компиляции:

```
Gen<int> strOb = new Gen<int>(53); // Ошибка: нельзя использовать
                                   // простой тип!
```

Очевидно, что невозможность подобной передачи простых типов не является серьезным ограничением, поскольку всегда имеется возможность использовать объектные оболочки для инкапсуляции значений (как это сделано в предыдущем примере). Кроме того, механизм автоупаковки и автораспаковки Java делает использование оболочек прозрачным.

Различение обобщений по аргументам типа

Ключом к пониманию обобщений является тот факт, что ссылки на разные специфические версии одного и того же обобщенного типа несовместимы между собой. Так, наличие в предыдущем примере следующей строки кода привело бы к ошибке во время компиляции:

```
iOb = strOb; // Ошибка!
```

Несмотря на то что обе переменные, iOb и strOb, относятся к типу Gen<T>, они ссылаются на объекты разного типа, поскольку в их объявлениях указаны разные аргументы типа. Это и есть частный пример той безопасности типов, которая обеспечивается использованием обобщенных типов, способствующим предотвращению возможных ошибок.

Обобщенный класс с двумя параметрами типа

Обобщенные типы допускают объявление нескольких параметров типа. Параметры задаются в виде списка элементов, разделенных запятыми. В качестве примера ниже приведена переработанная версия класса TwoGen, в которой определены два параметра типа.

```
// Простой обобщенный класс с двумя параметрами типа: T и V
class TwoGen<T, V> { ◄——— Использование двух параметров типа
   T ob1;
   V ob2;

   // Передать конструктору класса ссылки на объекты типов T и V
   TwoGen(T o1, V o2) {
```

```
        ob1 = o1;
        ob2 = o2;
    }

    // Отобразить типы T и V
    void showTypes() {
        System.out.println("Тип T - это " + ob1.getClass().getName());
        System.out.println("Тип V - это " + ob2.getClass().getName());
    }

    T getob1() {
        return ob1;
    }

    V getob2() {
        return ob2;
    }
}

// Демонстрация класса TwoGen
class SimpGen {
    public static void main(String args[]) {
        TwoGen<Integer, String> tgObj =
                new TwoGen<Integer, String>(88, "Обобщения");

        // Отобразить типы
        tgObj.showTypes();

        // Получить и отобразить значения
        int v = tgObj.getob1();
        System.out.println("значение: " + v);

        String str = tgObj.getob2();
        System.out.println("значение: " + str);
    }
}
```

Передача типа Integer параметру T и типа String параметру V

В результате выполнения этой программы будет получен следующий результат.

```
Тип T - это java.lang.Integer
Тип V - это java.lang.String
значение: 88
значение: Обобщения
```

Обратите внимание на приведенное ниже объявление класса TwoGen.

```
class TwoGen<T, V> {
```

В нем определены два параметра типа, T и V, разделенные запятыми. А поскольку в этом классе используются два параметра типа, то при создании объекта на его основе следует указывать оба аргумента типа:

```
TwoGen<Integer, String> tgObj =
        new TwoGen<Integer, String>(88, "Обобщения");
```

В данном случае тип Integer передается в качестве параметра типа T, а тип String — в качестве параметра типа V. И хотя в этом примере типы аргументов отличаются, они могут и совпадать. Например, следующая строка кода вполне допустима:

```
TwoGen<String, String> x = new TwoGen<String, String>("A", "B");
```

В данном случае в качестве обоих параметров типа T и V передается один и тот же тип String. Очевидно, что если типы аргументов совпадают, то определять два параметра типа в обобщенном классе не нужно.

Общая форма обобщенного класса

Синтаксис, представленный в предыдущих примерах, можно обобщить. Ниже приведена общая форма объявления обобщенного класса.

```
class имя_класса<список_параметров_типа> { // ...
```

А вот как выглядит синтаксис объявления ссылки на обобщенный класс:

```
имя_класса<список_аргументов_типа> имя_переменной =
   new имя_класса<список_аргументов_типа>
      (список_аргументов_конструктора);
```

Ограниченные типы

В предыдущих примерах параметры типа могли заменяться любым типом класса. Такая подстановка годится для многих целей, но иногда полезно ограничить допустимый ряд типов, передаваемых в качестве параметра типа. Допустим, требуется создать обобщенный класс для хранения числовых значений и выполнения над ними различных математических операций, включая получение обратной величины или извлечение дробной части. Допустим также, что в этом классе предполагается выполнение математических операций над данными любых числовых типов: как целочисленных, так и с плавающей точкой. В таком случае будет вполне логично указывать числовой тип данных обобщенно, т.е. с помощью параметра типа. Для создания такого класса можно было бы написать примерно такой код.

```
// Класс NumericFns как пример неудачной попытки создать
// обобщенный класс для выполнения различных математических
// операций, включая получение обратной величины или
// извлечение дробной части числовых значений любого типа.
class NumericFns<T> {
   T num;

   // Передать конструктору ссылку на числовой объект
   NumericFns(T n) {
      num = n;
   }
```

```
    // Вернуть обратную величину
    double reciprocal() {
       return 1 / num.doubleValue(); // Ошибка!
    }

    // Вернуть дробную часть
    double fraction() {
       return num.doubleValue() - num.intValue(); // Ошибка!
    }

    // ...
}
```

К сожалению, класс `NumericFns` в том виде, в каком он приведен выше, не компилируется, так как оба метода, определенные в этом классе, содержат программную ошибку. Рассмотрим сначала метод `reciprocal()`, который пытается вернуть величину, обратную его параметру `num`. Для этого нужно разделить 1 на значение переменной `num`, которое определяется при вызове метода `doubleValue()`. Этот метод возвращает версию `double` числового объекта, хранящегося в переменной `num`. Как известно, все числовые классы, в том числе `Integer` и `Double`, являются подклассами, производными от класса `Number`, в котором определен метод `doubleValue()`, что делает его доступным для всех классов оболочек числовых типов. Но компилятору неизвестно, что объекты класса `NumericFns` предполагается создавать только для числовых типов данных. Поэтому при попытке скомпилировать класс `NumericFns` возникает ошибка, а соответствующее сообщение уведомит вас о том, что метод `doubleValue()` неизвестен. Аналогичная ошибка возникает дважды при компиляции метода `fraction()`, где вызываются методы `doubleValue()` и `intValue()`. Вызов любого из этих методов также будет сопровождаться сообщением компилятора о том, что они неизвестны. Чтобы разрешить данную проблему, нужно каким-то образом сообщить компилятору, что в качестве параметра типа `T` предполагается использовать только числовые типы. И нужно еще убедиться, что в действительности передаются *только* эти типы данных.

Для подобных случаев в Java предусмотрены *ограниченные типы*. При указании параметра типа можно задать верхнюю границу, объявив суперкласс, который должны наследовать все аргументы типа. Это делается с помощью ключевого слова `extends`:

```
<T extends суперкласс>
```

Это объявление сообщает компилятору о том, что параметр типа *T* может быть заменен только *суперклассом* или его *подклассами*. Таким образом, *суперкласс* определяет верхнюю границу в иерархии классов Java.

С помощью ограниченных типов можно устранить программные ошибки в классе `NumericFns`. Для этого следует указать верхнюю границу, как показано ниже.

```
// В этой версии класса NumericFns аргументом типа,
// заменяющим параметр типа T, должен стать класс Number
// или производный от него подкласс, как показано ниже.
class NumericFns<T extends Number> { ◄—— В данном случае аргументом типа должен
                                          быть либо number, либо подкласс Number
   T num;

   // Передать конструктору ссылку на числовой объект
   NumericFns(T n) {
      num = n;
   }

   // Вернуть обратную величину
   double reciprocal() {
      return 1 / num.doubleValue();
   }

   // Вернуть дробную часть
   double fraction() {
      return num.doubleValue() - num.intValue();
   }

   // ...
}

// Демонстрация класса NumericFns
class BoundsDemo {
   public static void main(String args[]) {

   NumericFns<Integer> iOb = ◄—— Допустимо, потому что Integer
                                 является подклассом Number
                        new NumericFns<Integer>(5);

   System.out.println("Обратная величина iOb - " + iOb.reciprocal());
   System.out.println("Дробная часть iOb - " + iOb.fraction());

   System.out.println();

   // Применение класса Double также допустимо.
   NumericFns<Double> dOb = ◄—— Тип Double также допустим
                        new NumericFns<Double>(5.25);

   System.out.println("Обратная величина dOb - " + dOb.reciprocal());
   System.out.println("Дробная часть dOb - " + dOb.fraction());

   // Следующая строка кода не будет компилироваться, так как
   // класс String не является производным от класса Number.
// NumericFns<String> strOb = new NumericFns<String>("Ошибка"); ◄—┐
   }                                                               |
}                                           Тип String недопустим,
                                            так как он не является
                                            подклассом Number
```

Ниже приведен результат выполнения данной программы.

```
Обратная величина iOb is 0.2
Дробная часть iOb is 0.0

Обратная величина dOb - 0.19047619047619047
Дробная часть dOb - 0.25
```

Как видите, для объявления класса NumericFns в данном примере используется следующая строка кода:

```
class NumericFns<T extends Number> {
```

Теперь тип T ограничен классом Number, а следовательно, компилятору Java известно, что для всех объектов типа T доступен метод doubleValue(), а также другие методы, определенные в классе Number. Это не только уже само по себе дает немалые преимущества, но и предотвращает создание объектов класса NumericFns для нечисловых типов. Если вы удалите комментарии из строки кода в конце программы и попытаетесь скомпилировать ее, то компилятор выдаст сообщение об ошибке, поскольку класс String не является подклассом, производным от класса Number.

Ограниченные типы особенно полезны в тех случаях, когда нужно обеспечить совместимость одного параметра типа с другим. Рассмотрим в качестве примера представленный ниже класс Pair. В нем хранятся два объекта, которые должны быть совместимы друг с другом.

```
class Pair<T, V extends T>{   <——— Тип V должен совпадать с типом T
                                   или быть его подклассом
   T first;
   V second;

   Pair(T a, V b) {
      first = a;
      second = b;
   }

   // ...
}
```

В классе Pair определены два параметра типа, T и V, причем тип V расширяет тип T. Это означает, что тип V должен быть либо того же типа, что и T, либо его подклассом. Благодаря такому объявлению гарантируется, что два параметра типа, передаваемые конструктору класса Pair, будут совместимы. Например, приведенные ниже строки кода корректны.

```
// Эта строка кода верна, поскольку T и V имеют тип Integer
Pair<Integer, Integer> x = new Pair<Integer, Integer>(1, 2);

// И эта строка кода верна, так как Integer - подкласс Number
Pair<Number, Integer> y = new Pair<Number, Integer>(10.4, 12);
```

А следующий фрагмент кода содержит ошибку.

```
// Эта строка кода недопустима, так как String не является
// подклассом Number
   Pair<Number, String> z = new Pair<Number, String>(10.4, "12");
```

В данном случае класс `String` не является производным от класса `Number`, что нарушает условие, указанное в объявлении класса `Pair`.

Использование шаблонов аргументов

Безопасность типов — вещь полезная, но иногда она может мешать созданию конструкций, идеальных во всех других отношениях. Допустим, требуется реализовать метод `absEqual()`, возвращающий значение `true` в том случае, если два объекта рассматриваемого ранее класса `NumericFns` содержат одинаковые абсолютные значения. Допустим также, что этот метод должен оперировать любыми типами числовых данных, которые могут храниться в сравниваемых объектах. Так, если один объект содержит значение 1,25 типа `Double`, а другой — значение 1,25 типа `Float`, то метод `absEqual()` должен возвращать логическое значение `true`. Один из способов реализации метода `absEqual()` состоит в том, чтобы передавать этому методу параметр типа `NumericFns`, а затем сравнивать его абсолютное значение с абсолютным значением текущего объекта и возвращать значение `true`, если эти значения совпадают. Например, вызов метода `absEqual()` может выглядеть следующим образом.

```
NumericFns<Double> dOb = new NumericFns<Double>(1.25);
NumericFns<Float> fOb = new NumericFns<Float>(-1.25);

if(dOb.absEqual(fOb))
   System.out.println("Абсолютные значения совпадают");
else
   System.out.println("Абсолютные значения отличаются");
```

На первый взгляд может показаться, что при выполнении метода `absEqual()` не должно возникнуть никаких затруднений, но это совсем не так. Проблемы начнутся при первой же попытке объявить параметр типа `NumericFns`. Каким он должен быть? Казалось бы, подходящим должно быть следующее решение, где `T` указывается в качестве параметра типа.

```
// Это не будет работать!
// Определяем, будут ли совпадать абсолютные значения
// двух объектов.
boolean absEqual(NumericFns<T> ob) {
   if(Math.abs(num.doubleValue()) ==
      Math.abs(ob.num.doubleValue()) return true;

   return false;
}
```

В данном случае для определения абсолютного значения каждого числа используется стандартный метод Math.abs(). Далее выполняется сравнение полученных значений. Проблема состоит в том, что приведенное выше решение будет работать только тогда, когда объект класса NumericFns, передаваемый в качестве параметра, имеет тот же тип, что и текущий объект. Например, если текущий объект относится к типу NumericFns<Integer>, то параметр ob также должен быть типа NumericFns<Integer>, а следовательно, сравнить текущий объект с объектом типа NumericFns<Double> не удастся. Таким образом, выбранное решение не является обобщенным.

Для того чтобы создать обобщенный метод absEqual(), следует использовать еще одно средство обобщений: ***шаблон аргумента***. Шаблон обозначается метасимволом ?, которому соответствует неизвестный тип данных. Используя этот метасимвол, можно переписать метод absEqual() в следующем виде:

```
// Проверить равенство абсолютных значений двух объектов
boolean absEqual(NumericFns<?> ob) ◄——— Обратите внимание на метасимвол
   if(Math.abs(num.doubleValue()) ==
      Math.abs(ob.num.doubleValue()) return true;

   return false;
}
```

В данном случае выражение NumericFns<?> соответствует любому типу объекта из класса NumericFns, что позволяет сравнивать абсолютные значения в двух произвольных объектах класса NumericFns. Ниже приведен пример программы, демонстрирующий использование шаблона аргумента.

```
// Использование шаблона аргумента
class NumericFns<T extends Number> {
   T num;

   // Передать конструктору ссылку на числовой объект
   NumericFns(T n) {
      num = n;
   }

   // Вернуть обратную величину
   double reciprocal() {
      return 1 / num.doubleValue();
   }

   // Вернуть дробную часть
   double fraction() {
      return num.doubleValue() - num.intValue();
   }

   // Проверить равенство абсолютных значений двух объектов
   boolean absEqual(NumericFns<?> ob) {
      if(Math.abs(num.doubleValue()) ==
         Math.abs(ob.num.doubleValue())) return true;
```

```
      return false;
   }

   // ...
}

// Демонстрация использования шаблона аргумента
class WildcardDemo {
   public static void main(String args[]) {

      NumericFns<Integer> iOb = new NumericFns<Integer>(6);
      NumericFns<Double> dOb = new NumericFns<Double>(-6.0);
      NumericFns<Long> lOb = new NumericFns<Long>(5L);

      System.out.println("Сравнение iOb и dOb");
      if(iOb.absEqual(dOb))  ◄── В этом вызове метода тип аргумента-
                                 шаблона совпадает с типом Double
         System.out.println("Абсолютные значения совпадают.");
      else
         System.out.println("Абсолютные значения отличаются.");

      System.out.println();

      System.out.println("Сравнение iOb и lOb.");
      if(iOb.absEqual(lOb))  ◄── В этом вызове метода тип аргумента-
                                 шаблона совпадает с типом Long
         System.out.println("Абсолютные значения совпадают.");
      else
         System.out.println("Абсолютные значения отличаются.");

   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
Сравнение iOb and dOb
Абсолютные значения совпадают.

Сравнение iOb и lOb.
Абсолютные значения отличаются.
```

Обратите внимание на два следующих вызова метода `absEqual()`.

```
if(iOb.absEqual(dOb))

if(iOb.absEqual(lOb))
```

В первом вызове `iOb` — объект типа `NumericFns<Integer>`, а `dOb` — объект типа `NumericFns<Double>`. Однако использование шаблона в объявлении метода `absEqual()` позволило вызвать этот метод для объекта `iOb`, указав объект `dOb` в качестве аргумента. То же самое относится и к другому вызову, в котором методу передается объект типа `NumericFns<Long>`.

И последнее замечание: не следует забывать, что шаблоны аргументов не влияют на тип создаваемого объекта в классе NumericFns. Для этой цели служит спецификация extends в объявлении класса NumericFns. Шаблон лишь указывает на соответствие любому допустимому объекту класса NumericFns.

Ограниченные шаблоны

Шаблоны аргументов можно ограничивать в основном так же, как и параметры типов. Ограниченные шаблоны особенно полезны при создании методов, которые должны оперировать только объектами подклассов определенного суперкласса. Чтобы разобраться, почему это так, обратимся к простому примеру. Допустим, имеется следующий ряд классов, где класс A расширяется классами B и C, но не D.

```
class A {
   // ...
}

class B extends A {
   // ...
}

class C extends A {
   // ...
}

// Обратите внимание на то, что D не является подклассом A
class D {
   // ...
}
```

Рассмотрим простой обобщенный класс.

```
// Простой обобщенный класс
class Gen<T> {
   T ob;

   Gen(T o) {
      ob = o;
   }
}
```

В классе Gen предусмотрен один параметр типа, который определяет тип объекта, хранящегося в переменной ob. Как видите, на тип T не налагается никаких ограничений. Следовательно, параметр типа T может обозначать любой класс.

А теперь допустим, что требуется создать метод, который получает аргумент любого типа, соответствующего объекту класса Gen, при условии, что в качестве параметра типа этого объекта указывается класс A или его подклассы. Иными словами, требуется создать метод, который оперирует только объектами типа

`Gen<`*тип*`>`, где *тип* — это класс `A` или его подклассы. Для этой цели нужно воспользоваться ограниченным шаблоном аргумента. Ниже приведен пример объявления метода `test()`, которому в качестве аргумента может быть передан только объект класса `Gen`, чей параметр типа обязан соответствовать классу `A` или его подклассам.

```
// Здесь шаблон ? устанавливает соответствие
// классу A или его подклассам
static void test(Gen<? extends A> o) {
   // ...
}
```

Следующий пример класса демонстрирует типы объектов класса `Gen`, которые могут быть переданы методу `test()`.

```
class UseBoundedWildcard {
   // Здесь знак ? устанавливает соответствие
   // классу A или производным от него подклассам.
   static void test(Gen<? extends A> o) {  ◄── Использование ограниченного шаблона
      // ...
   }

   public static void main(String args[]) {
      A a = new A();
      B b = new B();
      C c = new C();
      D d = new D();

      Gen<A> w = new Gen<A>(a);
      Gen<B> w2 = new Gen<B>(b);
      Gen<C> w3 = new Gen<C>(c);
      Gen<D> w4 = new Gen<D>(d);

      // Эти вызовы метода test() допустимы
      test(w);
      test(w2);  ──── Эти вызовы метода test() допустимы, так как
      test(w3);       объекты w, w2 и w3 относятся к подклассам A

      // А этот вызов метода test() недопустим, так как
      // объект w4 не относится к подклассу A
//    test(w4); // Ошибка! ◄── А этот вызов недопустим, поскольку
   }                           объект w4 не является подклассом A
}
```

В методе `main()` создаются объекты классов `A`, `B`, `C` и `D`, которые затем используются для создания четырех объектов класса `Gen` (по одному на каждый тип). После этого метод `test()` вызывается четыре раза, причем последний его вызов закомментирован. Первые три вызова вполне допустимы, поскольку `w`, `w2` и `w3` являются объектами класса `Gen`, типы которых определяются классом `A` или производными от него классами. Последний вызов метода `test()` недопустим, потому что `w4` — это объект класса `D`, не являющегося производным от

класса `A`. Следовательно, ограниченный шаблон аргумента в методе `test()` не позволяет передавать ему объект `w4` в качестве параметра.

В общем случае для настройки верхней границы шаблона аргумента используется выражение следующего вида:

```
<? extends суперкласс >
```

где после ключевого слова `extends` указывается *суперкласс*, т.е. имя класса, определяющего верхнюю границу, включая и его самого. Это означает, что в качестве аргумента допускается указывать не только подклассы данного класса, но и сам этот класс.

По необходимости можно указать также нижнюю границу. Для этого используется ключевое слово `super`, указываемое в следующей общей форме:

```
<? super подкласс >
```

В данном случае в качестве аргумента допускается использовать только суперклассы, от которых наследуется подкласс, включая его самого.

Спросим у эксперта

ВОПРОС. Можно ли привести один экземпляр обобщенного класса к другому?

ОТВЕТ. Да, можно. Но только в том случае, если типы обоих классов совместимы и их аргументы типа совпадают. Рассмотрим в качестве примера обобщенный класс `Gen`:

```
class Gen<T> { // ...
```

Далее допустим, что переменная `x` объявлена следующим образом:

```
Gen<Integer> x = new Gen<Integer>();
```

В этом случае может быть выполнено следующее приведение типов, поскольку переменная `x` — это экземпляр класса `Gen<Integer>`:

```
(Gen<Integer>) x // Допустимо
```

А следующее приведение типов не может быть выполнено, поскольку переменная `x` не является экземпляром класса `Gen<Long>`:

```
(Gen<Long>) x // Недопустимо
```

Обобщенные методы

Как было продемонстрировано в предыдущих примерах, методы в обобщенных классах могут использовать параметр типа своего класса и следовательно, автоматически становятся обобщенными относительно параметра типа. Однако можно объявить обобщенный метод, который сам по себе использует параметры типа. Более того, такой метод может быть объявлен в обычном, а не обобщенном классе.

Ниже приведен пример программы, в которой объявляется класс GenericMethodDemo, не являющийся обобщенным. В этом классе объявляется статический обобщенный метод arraysEqual(), в котором определяется, содержатся ли в двух массивах одинаковые элементы, расположенные в том же самом порядке. Такой метод можно использовать для сравнения любых двух массивов с одинаковыми или совместимыми типами, а сами элементы массивов допускают их сравнение.

```
// Пример простого обобщенного метода
class GenericMethodDemo {

   // Определить, совпадает ли содержимое двух массивов
   static <T extends Comparable<T>, V extends T> boolean
      arraysEqual(T[] x, V[] y) { ◄──── Обобщенный метод
      // Массивы, имеющие разную длину, не могут быть одинаковыми
      if(x.length != y.length) return false;

      for(int i=0; i < x.length; i++)
         if(!x[i].equals(y[i])) return false; // массивы отличаются

      return true; // содержимое массивов совпадает
   }

   public static void main(String args[]) {

      Integer nums[] = { 1, 2, 3, 4, 5 };
      Integer nums2[] = { 1, 2, 3, 4, 5 };
      Integer nums3[] = { 1, 2, 7, 4, 5 };
      Integer nums4[] = { 1, 2, 7, 4, 5, 6 };

      if(arraysEqual(nums, nums)) ◄──── Аргументы типа T и V определяются неявно при вызове метода
         System.out.println("nums эквивалентен nums");

      if(arraysEqual(nums, nums2))
         System.out.println("nums эквивалентен nums2");

      if(arraysEqual(nums, nums3))
         System.out.println("nums эквивалентен nums3");

      if(arraysEqual(nums, nums4))
         System.out.println("nums эквивалентен nums4");

      // создать массив типа Double
      Double dvals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

      // Следующая строка не будет скомпилирована, так как
      // типы массивов nums и dvals не совпадают
//    if(arraysEqual(nums, dvals))
//       System.out.println("nums эквивалентен dvals");
   }
}
```

Результат выполнения данной программы выглядит следующим образом:

```
nums эквивалентен nums
nums эквивалентен nums2
```

Рассмотрим подробнее исходный код метода `arraysEqual()`. Прежде всего, взгляните на его объявление.

```
static <T extends Comparable<T>, V extends T> boolean
   arraysEqual(T[] x, V[] y) {
```

Параметры типа объявляются *перед* возвращаемым типом. Также обратите внимание на то, что `T` наследует интерфейс `Comparable<T>`. Интерфейс `Comparable` определен в пакете `java.lang`. Класс, реализующий интерфейс `Comparable`, определяет упорядочиваемые объекты. Таким образом, установление `Comparable` в качестве верхней границы допустимых типов гарантирует, что метод `arraysEqual()` можно использовать только в отношении объектов, допускающих сравнение. Интерфейс `Comparable` — обобщенный, и его параметр типа задает тип сравниваемых объектов. (О создании обобщенных интерфейсов речь пойдет позже.) Заметьте, что тип `V` ограничен сверху типом `T`. Следовательно, тип `V` должен либо совпадать с типом `T`, либо быть его подклассом. В силу этого метод `arrayEquals()` может вызываться лишь с аргументами, которые можно сравнивать между собой. Кроме того, этот метод объявлен как статический и поэтому вызывается без привязки к какому-либо объекту. Однако следует понимать, что обобщенные методы могут быть как статическими, так и нестатическими. Никаких ограничений в этом отношении не существует.

А теперь проанализируем, каким образом метод `arraysEqual()` вызывается в методе `main()`. Для этого используется обычный синтаксис, не требующий указания аргументов типа. Дело в том, что типы аргументов распознаются автоматически, соответственно определяя типы `T` и `V`. Например, в первом вызове

```
if(arraysEqual(nums, nums))
```

типом первого аргумента является `Integer`, который и подставляется вместо типа `T`. Таким же является и тип второго аргумента, а следовательно, тип параметра `V` также заменяется на `Integer`. Таким образом, выражение для вызова метода `arraysEqual()` составлено корректно, и сравнение массивов между собой может быть выполнено.

Обратите внимание на следующие строки, помещенные в комментарий.

```
// if(arraysEqual(nums, dvals))
//   System.out.println("nums эквивалентен dvals");
```

Если удалить в них символы комментариев и попытаться скомпилировать программу, то компилятор выдаст сообщение об ошибке. Дело в том, что верхней границей для типа параметра `V` является тип параметра `T`. Этот тип указывается после ключевого слова `extends`, т.е. тип параметра `V` может быть таким же, как и у параметра `T`, или быть его подклассом. В данном случае типом первого аргумента метода является `Integer`, заменяющий тип параметра `T`, тогда

как тип второго аргумента — Double, не являющийся подклассом Integer. Таким образом, вызов метода arraysEqual() оказывается недопустимым, что и приводит к ошибке во время компиляции.

Синтаксис объявления метода arraysEqual() может быть обобщен. Ниже приведена общая форма объявления обобщенного метода.

```
<список_параметров_типа> возвращаемый_тип
                          имя_метода(список параметров) {// ...
```

Во всех случаях параметры типа разделяются в списке запятыми. В объявлении обобщенного метода этот список предшествует объявлению возвращаемого типа.

Обобщенные конструкторы

Конструктор может быть обобщенным, даже если сам класс не является таковым. Например, в приведенной ниже программе класс Summation не является обобщенным, но в нем используется обобщенный конструктор.

```
// Использование обобщенного конструктора
class Summation {
   private int sum;

   <T extends Number> Summation(T arg) { ◄——— Обобщенный конструктор
      sum = 0;

      for(int i=0; i <= arg.intValue(); i++)
         sum += i;
   }

   int getSum() {
      return sum;
   }
}

class GenConsDemo {
   public static void main(String args[]) {
      Summation ob = new Summation(4.0);

      System.out.println("Сумма целых чисел от 0 до 4.0 равна " +
                         ob.getSum());
   }
}
```

В классе Summation вычисляется и инкапсулируется сумма всех чисел от 0 до *N*, причем значение *N* передается конструктору. Для конструктора Summation() указан параметр типа, ограниченный сверху классом Number, и поэтому объект типа Summation может быть создан с использованием любого числового типа, в том числе Integer, Float и Double. Независимо от того, какой числовой тип используется, соответствующее значение преобразуется в

тип Integer при вызове intValue(), после чего вычисляется требуемая сумма. Таким образом, класс Summation не обязательно объявлять обобщенным — достаточно сделать обобщенным только его конструктор.

Обобщенные интерфейсы

Как уже было показано на примере класса GenericMetodDemo, обобщенными могут быть не только классы и методы, но и интерфейсы. Использование в этом классе стандартного интерфейса Comparable<T> гарантировало возможность сравнения элементов двух массивов. Разумеется, вы также можете объявить собственный обобщенный интерфейс. Обобщенные интерфейсы объявляются аналогично тому, как объявляются обобщенные классы. В приведенном ниже примере программы создается обобщенный интерфейс Containment, который может быть реализован классами, хранящими одно или несколько значений. Кроме того, в этой программе объявляется метод contains(), позволяющий определить, содержится ли указанное значение в текущем объекте.

```
// Пример обобщенного интерфейса.

// Подразумевается, что класс, реализующий этот
// интерфейс, содержит одно или несколько значений
interface Containment<T> { ◄——— Обобщенный интерфейс
   // Метод contains() проверяет, содержится ли
   // некоторый элемент в объекте класса,
   // реализующего интерфейс Containment
   boolean contains(T o);
}

// Реализовать интерфейс Containment с помощью массива,
// предназначенного для хранения значений
class MyClass<T> implements Containment<T> { ◄——— Любой класс, реализующий
   T[] arrayRef;                                    обобщенный интерфейс,
                                                    также должен быть
                                                    обобщенным
   MyClass(T[] o) {
      arrayRef = o;
   }

   // Реализовать метод contains()
   public boolean contains(T o) {
      for(T x : arrayRef)
         if(x.equals(o)) return true;
      return false;
   }
}

class GenIFDemo {
   public static void main(String args[]) {
      Integer x[] = { 1, 2, 3 };
```

```
      MyClass<Integer> ob = new MyClass<Integer>(x);

      if(ob.contains(2))
         System.out.println("2 содержится в ob");
      else
         System.out.println("2 НЕ содержится в ob");

      if(ob.contains(5))
         System.out.println("5 содержится в ob");
      else
         System.out.println("5 НЕ содержится в ob");

      // Следующие строки кода недопустимы, так как объект ob
      // является вариантом реализации интерфейса Containment для
      // типа Integer, а значение 9.25 относится к типу Double
//    if(ob.contains(9.25)) // Недопустимо!
//       System.out.println("9.25 не содержится в ob");
   }
}
```

В результате выполнения этой программы будет получен следующий результат.

```
2 содержится в ob
5 Не содержится в ob
```

Большая часть исходного кода этой программы совершенно понятна, однако не помешает сделать пару замечаний. Прежде всего, обратите внимание на то, как объявляется интерфейс Containment:

```
interface Containment<T> {
```

Нетрудно заметить, что это объявление напоминает объявление обобщенного класса. В данном случае параметр типа T задает тип объектов содержимого.

Интерфейс Containment реализуется с помощью класса MyClass, объявление которого приведено ниже.

```
class MyClass<T> implements Containment<T> {
```

Если класс реализует обобщенный интерфейс, то он также должен быть обобщенным. В нем должен быть объявлен как минимум тот же параметр типа, что и в объявлении интерфейса. Например, такой вариант объявления класса MyClass недопустим:

```
class MyClass implements Containment<T> { // Ошибка!
```

В данном случае ошибка состоит в том, что в классе MyClass не объявлен параметр типа, а это означает, что передать параметр типа интерфейсу Containment невозможно. Если идентификатор T останется неизвестным, компилятор выдаст сообщение об ошибке. Класс, реализующий обобщенный интерфейс, может не быть обобщенным только в одном случае: если при объявлении класса для интерфейса указывается конкретный тип:

```
class MyClass implements Containment<Double> { // Допустимо
```

Вас теперь вряд ли удивит, что один или несколько параметров типа, определяемых обобщенным интерфейсом, могут быть ограниченными. Это позволяет указать, какие именно типы данных допустимы для интерфейса. Например, если нужно ограничить применимость интерфейса `Containment` числовыми типами, то его можно объявить следующим образом:

```
interface Containment<T extends Number> {
```

Теперь любой класс, реализующий интерфейс `Containment`, должен передавать ему значение типа, удовлетворяющее тем же ограничениям. Например, класс `MyClass`, реализующий данный интерфейс, должен объявляться следующим образом:

```
class MyClass<T extends Number> implements Containment<T> {
```

Обратите внимание на то, как параметр типа `T` объявляется в классе `MyClass`, а затем передается интерфейсу `Containment`. На этот раз интерфейсу `Containment` требуется тип, расширяющий тип `Number`, поэтому в классе `MyClass`, реализующем этот интерфейс, должны быть указаны соответствующие ограничения. Если верхняя граница задана в объявлении класса, то нет никакой необходимости указывать ее еще раз после ключевого слова `implements`. Если же вы попытаетесь это сделать, то компилятор выдаст сообщение об ошибке. Например, следующее выражение некорректно и не будет скомпилировано.

```
// Ошибка!
class MyClass<T extends Number>
     implements Containment<T extends Number> {
```

Коль скоро параметр типа задан, он просто передается интерфейсу без дальнейших видоизменений.

Ниже приведен синтаксис объявления обобщенного интерфейса.

```
interface имя_интерфейса<список_параметров_типа> { // ...
```

где *список_параметров_типа* содержит список параметров, разделенных запятыми. При реализации обобщенного интерфейса в объявлении класса также должны быть указаны параметры типа. Общая форма объявления класса, реализующего обобщенный интерфейс, приведена ниже.

```
class имя_класса<список_параметров_типа>
     implements имя_интерфейса<список_параметров_типа> {
```

Упражнение 13.1 Создание обобщенного класса очереди

```
IGenQ.java
QucueFullException.java
QueueEmptyException.java
GenQueue.java
GenQDemo.java
```

Одним из главных преимуществ обобщенных классов является возможность создания надежного кода, пригодного для повторного использования. Как уже упоминалось в начале главы, многие алгоритмы могут быть реализованы одинаково, независимо от типа данных. Например, очередь в равной степени пригодна для хранения целых чисел, строк, объектов типа `File` и других типов данных. Вместо того чтобы создавать

отдельный класс очереди для объектов каждого типа, можно разработать единое обобщенное решение, позволяющее работать с объектами любого типа. В итоге цикл проектирования, программирования, тестирования и отладки кода будет выполняться только один раз, и его не нужно будет проходить заново, когда потребуется организовать очередь для нового типа данных.

В этом упражнении вам предстоит видоизменить класс очереди, разработка которого была начата в упражнении 5.2, и придать ему окончательный вид. Проект включает обобщенный интерфейс, определяющий операции над очередью, два класса исключений и один вариант реализации — очередь фиксированного размера. Разумеется, вам ничто не помешает поэкспериментировать с другими разновидностями обобщенных очередей, например создать динамическую или циклическую очередь, следуя приведенным ниже рекомендациям.

Как и предыдущая версия очереди, реализованная в упражнении 9.1, исходный код, реализующий очередь в этом упражнении, будет организован в виде ряда отдельных файлов. С этой целью код интерфейса, исключений, реализации очереди фиксированного размера и программы, демонстрирующей очередь в действии, будет распределен по отдельным исходным файлам. Такая организация исходного кода отвечает подходу, принятому в работе над большинством реальных проектов. Поэтапное описание процесса создания программы приведено ниже.

1. Первым этапом создания обобщенной очереди станет формирование обобщенного интерфейса, описывающего две операции над очередью: размещение и извлечение объектов. Обобщенная версия интерфейса очереди будет называться `IGenQ`, и ее исходный код приведен ниже. Поместите этот код в файл `IGenQ.java`.

```
// Обобщенный интерфейс очереди
public interface IGenQ<T> {
   // Поместить элемент в очередь
   void put(T ch) throws QueueFullException;

   // Извлечь элемент из очереди
   T get() throws QueueEmptyException;
}
```

   Обратите внимание на то, что тип данных, предназначенных для хранения в очереди, определяется параметром типа `T`.

2. Создайте файлы `QueueFullException.java` и `QueueEmptyException.java` и введите в каждый из них исходный код одноименного класса.

```
// Исключение, указывающее на переполнение
class QueueFullException extends Exception {
   int size;

   QueueFullException(int s) { size = s; }

   public String toString() {
      return "\nОчередь заполнена. Максимальный размер очереди: " +
```

```
            size;
   }
}

// Исключение, указывающее на исчерпание очереди
class QueueEmptyException extends Exception {

   public String toString() {
      return "\nОчередь пуста";
   }
}
```

В этих классах инкапсулированы две ошибки, которые могут возникнуть в работе с очередью: попытка поместить элемент в заполненную очередь и попытка извлечь элемент из пустой очереди. Эти классы не являются обобщенными, поскольку они действуют одинаково, независимо от типа данных, хранящихся в очереди, и поэтому совпадают с теми, которые использовались в упражнении 9.1.

3. Создайте файл `GenQueue.java`. Введите в него приведенный ниже код, реализующий очередь фиксированного размера.

```
// Обобщенный класс, реализующий очередь фиксированного размера
class GenQueue<T> implements IGenQ<T> {
   private T q[];              // массив для хранения элементов
                               // очереди
   private int putloc, getloc; // индексы вставки и извлечения
                               // элементов очереди

   // Создание пустой очереди из заданного массива
   public GenQueue(T[] aRef) {
      q = aRef;
      putloc = getloc = 0;
   }

   // Поместить элемент в очередь
   public void put(T obj) throws QueueFullException {

      if(putloc==q.length)
         throw new QueueFullException(q.length);

      q[putloc++] = obj;
   }

   // Извлечь элемент из очереди
   public T get() throws QueueEmptyException {

      if(getloc == putloc)
         throw new QueueEmptyException();

      return q[getloc++];
   }
}
```

Класс `GenQueue` объявляется как обобщенный с параметром типа `T`. Этот параметр определяет тип данных, хранящихся в очереди. Обратите внимание на то, что параметр типа `T` также передается интерфейсу `IGenQ`.

Конструктору `GenQueue` передается ссылка на массив, используемый для хранения элементов очереди. Следовательно, чтобы создать объект класса `GenQueue`, необходимо сначала создать массив, тип которого совместим с типом объектов, сохраняемых в очереди, а его размер достаточен для размещения объектов в очереди.

Например, в следующих строках кода показано, как создать очередь для хранения строк:

```
String strArray[] = new String[10];
GenQueue<String> strQ = new GenQueue<String>(strArray);
```

4. **Создайте файл `GenQDemo.java` и введите в него приведенный ниже код, демонстрирующий работу обобщенной очереди.**

```
/*
   Упражнение 13.1.

   Демонстрация обобщенного класса очереди.
*/

class GenQDemo {
   public static void main(String args[]) {
      // Создать очередь для хранения целых чисел
      Integer iStore[] = new Integer[10];
      GenQueue<Integer> q = new GenQueue<Integer>(iStore);

      Integer iVal;

      System.out.println("Демонстрация очереди чисел типа Integer");
      try {
         for(int i=0; i < 5; i++) {
            System.out.println("Добавление " + i + " в очередь q");
            q.put(i); // добавить целочисленное значение в очередь q
         }
      }
      catch (QueueFullException exc) {
         System.out.println(exc);
      }
      System.out.println();

      try {
         for(int i=0; i < 5; i++) {
            System.out.print("Получение следующего числа
                              типа Integer из очереди q: ");
            iVal = q.get();
            System.out.println(iVal);
         }
      }
```

```
        catch (QueueEmptyException exc) {
           System.out.println(exc);
        }

        System.out.println();

        // Создать очередь для хранения чисел с плавающей точкой
        Double dStore[] = new Double[10];
        GenQueue<Double> q2 = new GenQueue<Double>(dStore);

        Double dVal;

        System.out.println("Демонстрация очереди чисел типа Double");
        try {
           for(int i=0; i < 5; i++) {
              System.out.println("Добавление " + (double)i/2 +
                                 " в очередь q2");
              q2.put((double)i/2); // ввести значение типа double
                                   // в очередь q2
           }
        }
        catch (QueueFullException exc) {
           System.out.println(exc);
        }
        System.out.println();

        try {
           for(int i=0; i < 5; i++) {
              System.out.print("Получение следующего числа типа
                                Double из очереди q2: ");
              dVal = q2.get();
              System.out.println(dVal);
           }
        }
        catch (QueueEmptyException exc) {
           System.out.println(exc);
        }
    }
}
```

5. Скомпилируйте программу и запустите на выполнение. В итоге на экране отобразится следующий результат.

```
Демонстрация очереди чисел типа Integer
Добавление 0 в очередь q.
Добавление 1 в очередь q.
Добавление 2 в очередь q.
Добавление 3 в очередь q.
Добавление 4 в очередь q.

Получение следующего числа типа Integer из очереди q: 0
Получение следующего числа типа Integer из очереди q: 1
```

```
Получение следующего числа типа Integer из очереди q: 2
Получение следующего числа типа Integer из очереди q: 3
Получение следующего числа типа Integer из очереди q: 4

Демонстрация очереди чисел типа Double
Добавление 0.0 в очередь q2.
Добавление 0.5 в очередь q2.
Добавление 1.0 в очередь q2.
Добавление 1.5 в очередь q2.
Добавление 2.0 в очередь q2.

Получение следующего числа типа Double из очереди q2: 0.0
Получение следующего числа типа Double из очереди q2: 0.5
Получение следующего числа типа Double из очереди q2: 1.0
Получение следующего числа типа Double из очереди q2: 1.5
Получение следующего числа типа Double из очереди q2: 2.0
```

6. Попытайтесь самостоятельно написать обобщенные версии классов `CircularQueue` и `DynQueue`, созданных в упражнении 8.1.

Базовые типы и унаследованный код

Поскольку в версиях Java, предшествующих JDK 5, поддержка обобщенных типов отсутствовала, необходимо было предпринять меры к тому, чтобы обеспечить совместимость новых программ с унаследованным кодом. По сути, возникла потребность в средствах, позволяющих унаследованному коду сохранить свою функциональность и при этом иметь возможность взаимодействовать с кодом, использующим обобщенные типы.

Чтобы облегчить адаптацию существующего кода к обобщениям, Java позволяет использовать обобщенные классы без указания аргументов типа. В результате для класса создается так называемый "сырой" (далее — базовый) тип. Базовые типы совместимы с унаследованным кодом, которому ничего не известно об обобщенных классах. Главный недостаток использования базовых типов заключается в том, что безопасность типов, обеспечиваемая обобщениями, при этом утрачивается.

Ниже приведен пример программы, демонстрирующей использование базового типа.

```
// Демонстрация использования базового типа
class Gen<T> {
   T ob; // объявить объект типа T

   // Передать конструктору ссылку на объект типа T
   Gen(T o) {
      ob = o;
   }
```

```
   // Вернуть объект ob
   T getob() {
      return ob;
   }
}

// Продемонстрировать использование базового типа
class RawDemo {
   public static void main(String args[]) {

      // Создать объект класса Gen для типа Integer
      Gen<Integer> iOb = new Gen<Integer>(88);

      // Создать объект класса Gen для типа String
      Gen<String> strOb = new Gen<String>("Тестирование обобщений");

      // Создать базовый объект класса Gen
      // и передать ему значение типа Double
      Gen raw = new Gen(new Double(98.6)); ◄—— Если аргумент типа не предоставляется,
                                               создается базовый тип

      // Здесь требуется приведение типов, так как тип неизвестен
      double d = (Double) raw.getob();
      System.out.println("значение: " + d);

      // Использование базового типа может привести
      // к исключениям времени выполнения. Соответствующие
      // примеры представлены ниже.

      // Следующее приведение типов вызывает ошибку
      // времени выполнения!
//    int i = (Integer) raw.getob(); // ошибка времени выполнения
                                                Безопасность использования
      // Это присваивание нарушает безопасность типов  базового типа не проверяется
      strOb = raw; // допустимо, но потенциально неверно ◄——┘
//    String str = strOb.getob(); // ошибка времени выполнения

      // Следующее присваивание также нарушает безопасность типов
      raw = iOb; // допустимо, но потенциально неверно
//    d = (Double) raw.getob(); // ошибка времени выполнения
   }
}
```

У этой программы имеется ряд интересных особенностей. Прежде всего, базовый тип обобщенного класса Gen создается в следующем объявлении:

```
Gen raw = new Gen(new Double(98.6));
```

В данном случае аргументы типа не указываются. В итоге создается объект класса Gen, тип T которого заменяется типом Object.

Базовые типы не обеспечивают безопасность типов. Переменной базового типа может быть присвоена ссылка на любой тип объекта класса Gen. Справедливо и обратное утверждение: переменной конкретного типа из класса Gen

может быть присвоена ссылка на объект класса Gen базового типа. Обе операции потенциально опасны, поскольку они действуют в обход механизма проверки типов, обязательной для обобщений.

Недостаточный уровень безопасности типов демонстрируют примеры в строках кода в конце данной программы, помещенных в комментарии. Рассмотрим их по отдельности. Сначала проанализируем следующую строку кода:

```
// int i = (Integer) raw.getob(); // ошибка времени выполнения
```

В этом операторе присваивания в объекте raw определяется значение переменной ob, которое приводится к типу Integer. Однако в объекте raw содержится не целое число, а значение типа Double. На стадии компиляции этот факт выявить невозможно, поскольку тип объекта raw неизвестен. Следовательно, ошибка возникнет на стадии выполнения программы.

В следующих строках кода ссылка на объект класса Gen базового типа присваивается переменной strOb (предназначенной для хранения ссылок на объекты типа Gen<String>).

```
   strOb = raw; // допустимо, но потенциально неверно
// String str = strOb.getob(); // ошибка времени выполнения
```

Само по себе присваивание синтаксически правильно, но все же сомнительно. Переменная strOb ссылается на объект типа Gen<String>, следовательно, она должна содержать ссылку на объект, содержащий значение типа String, но после присваивания объект, на который ссылается переменная strOb, содержит значение типа Double. Поэтому, когда во время выполнения программы предпринимается попытка присвоить переменной str содержимое объекта, на который ссылается переменная strOb, возникает ошибка. Причиной ошибки является то, что в этот момент переменная strOb ссылается на объект, содержащий значение типа Double. Таким образом, присваивание ссылки на объект базового типа переменной, ссылающейся на объект обобщенного типа, делается в обход механизма безопасности типов.

В следующих строках кода демонстрируется ситуация, обратная только что описанной.

```
   raw = iOb; // допустимо, но потенциально ошибочно
// d = (Double) raw.getob(); // ошибка при выполнении программы
```

В данном случае ссылка на объект обобщенного типа присваивается переменной базового типа. И это присваивание синтаксически правильно, но приводит к ошибке, возникающей во второй строке кода. В частности, переменная raw указывает на объект, содержащий значение типа Integer, но при приведении типов предполагается, что он содержит значение типа Double. Эту ошибку также нельзя выявить на стадии компиляции, так как она проявляется только на стадии выполнения программы.

В связи с тем, что использование базовых типов сопряжено с потенциальными рисками, в подобных случаях компилятор javac выводит так называемые *непроверенные предупреждения*, указывающие на возможность нарушения

безопасности типов. В рассматриваемой программе причиной таких предупреждений являются следующие строки кода.

```
Gen raw = new Gen(new Double(98.6));

strOb = raw; // допустимо, но потенциально неверно
```

В первой строке кода содержится обращение к конструктору класса Gen без указания аргумента типа, что приводит к выдаче компилятором соответствующего предупреждения. При компиляции второй строки предупреждающее сообщение возникнет из-за попытки присвоить переменной, ссылающейся на объект обобщенного типа, ссылки на объект базового типа.

На первый взгляд может показаться, что предупреждение об отсутствии проверки типов должна порождать и приведенная ниже строка кода, однако этого не происходит.

```
raw = iOb; // допустимо, но потенциально неверно
```

В данном случае компилятор не выдает никаких предупреждающих сообщений, потому что такое присваивание не вносит никакой дополнительной потери безопасности типов кроме той, которая уже была привнесена при создании переменной raw базового типа.

Из всего вышесказанного можно сделать следующий вывод: базовыми типами следует пользоваться весьма ограниченно и только в тех случаях, когда унаследованный код объединяется с новым, обобщенным кодом. Базовые типы — это лишь вспомогательное средство, необходимое для обеспечения совместимости с унаследованным кодом, и их использования во вновь создаваемом коде следует избегать.

Выведение типов с помощью ромбовидного оператора

Начиная с версии JDK 7 для создания экземпляров обобщенного типа предусмотрен сокращенный синтаксис. В качестве примера обратимся к классу TwoGen, представленному в начале этой главы. Ниже для удобства приведена часть его объявления. Обратите внимание на то, что в нем определяются два обобщенных типа данных.

```
class TwoGen<T, V> {
   T ob1;
   V ob2;

   // Передать конструктору ссылку на объект типа T
   TwoGen(T o1, V o2) {
      ob1 = o1;
      ob2 = o2;
   }
   // ...
}
```

В версиях Java, предшествующих JDK 7, для создания экземпляра класса TwoGen пришлось бы использовать примерно такой код.

```
TwoGen<Integer, String> tgOb =
   new TwoGen<Integer, String>(42, "testing");
```

Здесь аргументы типа (в данном случае Integer и String) указываются дважды: сначала при объявлении переменной tgOb, а затем при создании экземпляра класса TwoGen с помощью оператора new. С тех пор как обобщения были введены в версии JDK 5, подобная форма создания объектов обобщенного типа была обязательной для всех версий Java, предшествующих JDK 7. И хотя эта форма сама по себе верна, она более громоздка, чем это действительно требуется. Поскольку компилятору несложно самостоятельно определить типы аргументов в операторе new, дублирование этой информации излишне. Для разрешения подобной ситуации в версии JDK 7 предусмотрен специальный синтаксический элемент.

Версия JDK 7 позволяет переписать приведенное выше объявление в следующем виде:

```
TwoGen<Integer, String> tgOb = new TwoGen<>(42, "testing");
```

Обратите внимание на ту часть кода, в которой создается экземпляр объекта обобщенного типа. *Угловые скобки* (<>), называемые *ромбовидным оператором* и обозначающие пустой список аргументов типа, предписывают компилятору самостоятельно определить типы аргументов, требующиеся конструктору, исходя из контекста (так называемое *выведение типов*). Главное преимущество такого подхода состоит в том, что он позволяет существенно сократить размер неоправданно громоздких объявлений. Эта возможность оказывается особенно удобной при объявлении обобщенных типов, определяющих границы наследования в иерархии классов Java.

Приведенную выше форму объявления экземпляра класса можно обобщить. Для того чтобы компилятор автоматически определял (выводил) типы аргументов типа, необходимо использовать следующий синтаксис объявления обобщенной ссылки и создания экземпляра объекта обобщенного типа:

```
имя_класса <список_аргументов_типа> имя_переменной =
          new имя_класса<>(список_аргументов_конструктора);
```

В подобных случаях список аргументов типа в операторе new должен быть пустым.

Как правило, автоматическое выведение типов компилятором возможно и при передаче параметров методам. Так, если объявить в классе TwoGen следующий метод:

```
boolean isSame(TwoGen<T, V> o) {
   if(ob1 == o.ob1 && ob2 == o.ob2) return true;
   else return false;
}
```

то в JDK 7 будет вполне допустим вызов следующего вида.

```
if(tgOb.isSame(new TwoGen<>(42, "тестирование")))
   System.out.println("Совпадают");
```

В этом случае аргументы типа, которые должны передаваться методу `isSame()`, опускаются. Их типы могут быть автоматически определены компилятором, а следовательно, их повторное указание было бы излишним.

Возможность использования пустого списка аргументов типа появилась в версии JDK 7, а потому в более ранних версиях компилятора Java она недоступна. По этой причине в примерах программ, приводимых далее, будет использоваться прежний, несокращенный синтаксис объявления экземпляров обобщенных классов, который воспринимается любым компилятором Java, поддерживающим обобщения. Кроме того, несокращенный синтаксис позволяет яснее понять, какие именно объекты создаются, что делает примеры более наглядными и полезными. Однако использование синтаксиса выведения типов в собственных программах позволит вам значительно упростить объявления.

Очистка

Как правило, программисту не требуется знать все детали того, каким образом компилятор преобразует исходный код программы в объектный. Однако в случае обобщенных типов важно иметь хотя бы общее представление о процессе их преобразования. Это помогает лучше понять, почему обобщенные классы и методы действуют именно так, а не иначе, и почему иногда они ведут себя не совсем обычно. Поэтому ниже приведено краткое описание того, каким образом обобщенные типы реализуются в Java.

При реализации обобщенных типов в Java разработчикам пришлось учитывать важное ограничение, суть которого состоит в необходимости обеспечить совместимость с предыдущими версиями Java. Проще говоря, обобщенный код должен был быть совместимым с предыдущими версиями кода, разработанными до появления обобщенных типов. Таким образом, любые изменения в синтаксисе языка Java или механизме JVM не должны были нарушать работоспособность уже существующего кода. Поэтому для реализации обобщенных типов с учетом указанных ограничений был выбран механизм, получивший название *очистка*.

Механизм очистки работает следующим образом. При компиляции кода, написанного на языке Java, все сведения об обобщенных типах удаляются. Это означает, что параметры типа заменяются верхними границами их типа, а если границы не указаны, то их функции выполняет класс `Object`. После этого выполняется приведение типов, заданных аргументами типа. Подобная совместимость типов также контролируется компилятором. Это означает, что во время выполнения программы параметры типа просто не существуют. Этот механизм имеет отношение лишь к исходному коду.

Ошибки неоднозначности

Включение в язык обобщенных типов породило новый вид ошибок, от которых приходится защищаться, — *неоднозначность*. Ошибки неоднозначности возникают в тех случаях, когда процесс очистки порождает два различающихся на первый взгляд объявления обобщений, которые разрешаются в один и тот же очищенный тип, что приводит к возникновению конфликта. Рассмотрим пример, в котором используется перегрузка методов.

```
// Неоднозначность, вызванная очисткой перегруженных методов
class MyGenClass<T, V> {
   T ob1;
   V ob2;

   // ...

   // Эти два объявления перегруженных методов порождают
   // неоднозначность, и потому код не компилируется
   void set(T o) {   ◄──┐
      ob1 = o;          │
   }                    ├── Пара этих методов
                        │   порождает неоднозначность
   void set(V o) {   ◄──┘
      ob2 = o;
   }
}
```

Обратите внимание на то, что в классе `MyGenClass` объявлены два обобщенных типа: `T` и `V`. При этом предпринимается попытка перегрузить метод `set()` на основе параметров `T` и `V`. Это представляется вполне разумным, поскольку типы `T` и `V` — разные. Однако здесь возникают два затруднения, связанные с неоднозначностью.

Во-первых, в определении класса `MyGenClass` ничто не указывает на то, что типы `T` и `V` — действительно разные. Например, не является принципиальной ошибкой создание объекта типа `MyGenClass` так, как показано ниже.

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

В данном случае типы `T` и `V` будут заменены типом `String`. В результате оба варианта метода `set()` становятся совершенно одинаковыми, что, безусловно, является ошибкой.

Во-вторых, более серьезное затруднение возникает в связи с тем, что в результате очистки типов оба варианта метода `set()` преобразуются к следующему виду:

```
void set(Object o) { // ...
```

Таким образом, попытке перегрузить метод `set()` класса `MyGenClass` присуща неоднозначность. В данном случае вместо перегрузки методов вполне можно использовать два метода с различными именами.

Ограничения на использование обобщений

Существуют некоторые ограничения, которые следует учитывать, если вы используете обобщения. Эти ограничения касаются создания объектов параметров типа, статических членов, исключений и массивов. Ниже каждое из этих ограничений рассматривается по отдельности.

Невозможность создания экземпляров параметров типа

Создать экземпляр параметра типа невозможно. Рассмотрим в качестве примера следующий класс.

```
// Невозможно получить экземпляр типа T
class Gen<T> {
   T ob;

   Gen() {
      ob = new T(); // Недопустимо!!!
   }
}
```

В данном примере попытка получить экземпляр типа T приводит к ошибке. Причину этой ошибки понять нетрудно: компилятору ничего не известно о типе создаваемого объекта, поскольку тип T является заполнителем, информация о котором удаляется во время компиляции.

Ограничения статических членов класса

В статическом члене нельзя использовать параметры типа, объявленные в его классе. Так, все объявления статических членов в приведенном ниже классе недопустимы.

```
class Wrong<T> {
   // Неверно, поскольку невозможно создать
   // статическую переменную типа T
   static T ob;

   // Неверно, поскольку невозможно использовать
   // переменную типа T в статическом методе
   static T getob() {
      return ob;
}
```

Несмотря на наличие описанного выше ограничения, *допускается* объявлять обобщенные статические методы, которые определяют собственные параметры типа, как это было сделано ранее.

Ограничения обобщенных массивов

На массивы обобщенного типа накладываются два существенных ограничения. Во-первых, нельзя получить экземпляр массива, тип элементов которого определяется параметром типа. И во-вторых, нельзя создать массив

обобщенных ссылок на объекты конкретного типа. Оба этих ограничения демонстрируются в приведенном ниже примере.

```
// Обобщенные типы и массивы
class Gen<T extends Number> {
   T ob;

   T vals[]; // допустимо

   Gen(T o, T[] nums) {
      ob = o;

      // Следующее выражение недопустимо
//    vals = new T[10]; // невозможно создать массив типа T

      // Однако такой оператор допустим
      vals = nums; // присвоение ссылки на существующий
                   // массив допускается
   }
}

class GenArrays {
   public static void main(String args[]) {
      Integer n[] = { 1, 2, 3, 4, 5 };

      Gen<Integer> iOb = new Gen<Integer>(50, n);

      // Невозможно создать массив обобщенных ссылок
      // на объекты конкретного типа
//    Gen<Integer> gens[] = new Gen<Integer>[10]; // Ошибка!

      // Следующее выражение допустимо
      Gen<?> gens[] = new Gen<?>[10];
   }
}
```

Как показано в этой программе, ничто не мешает создать ссылку на массив типа `T`:

```
T vals[]; // допустимо
```

Однако получить экземпляр массива типа `T`, как показано в строке ниже, невозможно:

```
// vals = new T[10]; // невозможно создать массив типа T
```

В данном случае ограничение, налагаемое на массив типа `T`, состоит в том, что компилятору неизвестно, какого типа массив следует в действительности создавать. Но в то же время конструктору `Gen()` можно передать ссылку на массив совместимого типа при создании объекта, а также присвоить это значение переменной `vals`:

```
vals = nums; // присвоение ссылки на существующий
             // массив допускается
```

Это выражение работает, поскольку тип массива, передаваемого конструктору `Gen()` при создании объекта, известен и совпадает с типом `T`. В теле метода `main()` содержится выражение, демонстрирующее невозможность объявить массив обобщенных ссылок на объекты конкретного типа. Поэтому приведенная ниже строка кода не будет скомпилирована.

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Ошибка!
```

Ограничения обобщенных исключений

Обобщенный класс не может расширять класс `Throwable`. Это означает, что создавать обобщенные классы исключений невозможно.

Дальнейшее изучение обобщений

Как отмечалось в начале главы, приведенных в ней сведений вам будет достаточно для того, чтобы эффективно пользоваться обобщениями в программах на Java. Вместе с тем обобщения имеют немало особенностей, которые не были отражены в этой главе.

Читатели, которых заинтересовала данная тема, вероятно, захотят узнать больше о том влиянии, которое обобщения оказывают на иерархию классов, и, в частности, каким образом осуществляется сравнение типов во время выполнения, как переопределяются методы и пр. Все эти и многие другие вопросы, связанные с использованием обобщений, подробно освещены в книге *Java. Полное руководство, 10-е издание*.

Вопросы и упражнения для самопроверки

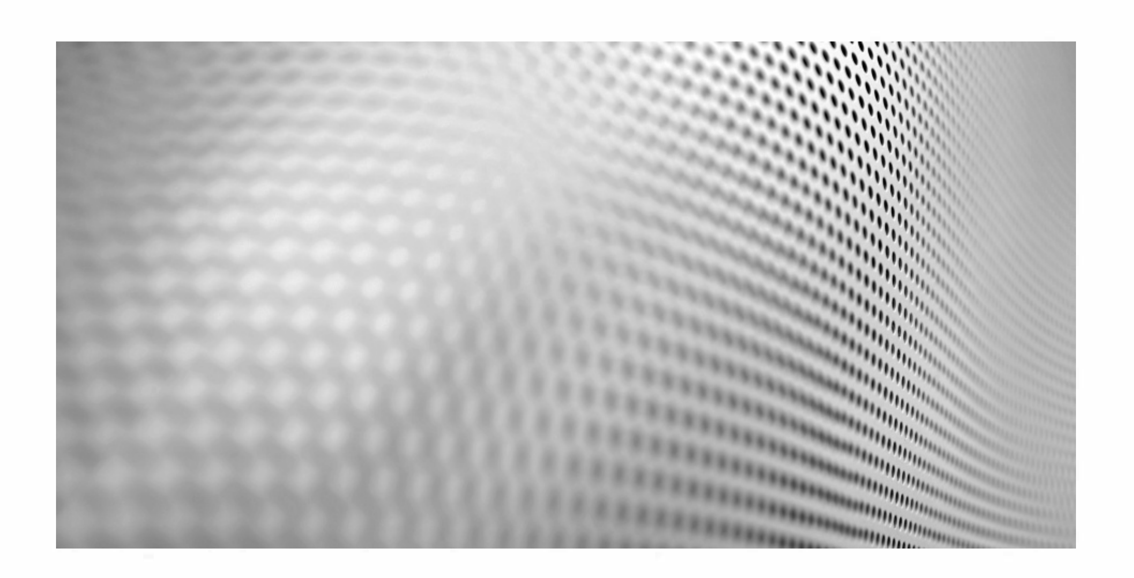
1. Обобщения очень важны, поскольку позволяют создавать код, который:
   а) обеспечивает безопасность типов;
   б) пригоден для повторного использования;
   в) отличается высокой надежностью;
   г) обладает всеми перечисленными выше свойствами.
2. Можно ли указывать простой тип в качестве аргумента типа?
3. Как объявить класс `FlightSched` с двумя параметрами типа?
4. Измените ваш ответ на вопрос 3 таким образом, чтобы второй параметр типа обозначал подкласс, производный от класса `Thread`.
5. Внесите изменения в класс `FlightSched` таким образом, чтобы второй параметр типа стал подклассом первого параметра типа.

6. Что обозначает знак `?` в обобщениях?
7. Может ли шаблон аргумента быть ограниченным?
8. У обобщенного метода `MyGen()` имеется один параметр типа, определяющий тип передаваемого ему аргумента. Этот метод возвращает также объект, тип которого соответствует параметру типа. Как должен быть объявлен метод `MyGen()`?
9. Допустим, обобщенный интерфейс объявлен так:

```
interface IGenIF<T, V extends T> { // ...
```

   Напишите объявление класса `MyClass`, который реализует интерфейс `IGenIF`.
10. Допустим, имеется обобщенный класс `Counter<T>`. Как создать объект его базового типа?
11. Существуют ли параметры типа на стадии выполнения программы?
12. Видоизмените ответ на вопрос 10 в упражнении для самопроверки из главы 9 таким образом, чтобы сделать класс обобщенным. Для этого создайте интерфейс стека `IGenStack`, объявив в нем обобщенные методы `push()` и `pop()`.
13. Что означает пара угловых скобок (`<>`)?
14. Как упростить приведенную ниже строку кода?

```
MyClass<Double,String> obj = new MyClass<Double,String>(1.1,"Привет");
```



Глава 14

Лямбда-выражения и ссылки на методы

В этой главе...

- Общая форма лямбда-выражений
- Определение функционального интерфейса
- Использование лямбда-выражений
- Использование блочных лямбда-выражений
- Использование обобщенных функциональных интерфейсов
- Захват переменных в лямбда-выражениях
- Генерация исключений в лямбда-выражениях
- Ссылки на методы
- Ссылки на конструкторы
- Стандартные функциональные интерфейсы, определенные в пакете `java.util.function`

Выпуск JDK 8 дополнил Java новым средством — *лямбда-выражениями*, которые способствовали значительному усилению выразительных возможностей языка. Лямбда-выражения не только вводят в язык новый синтаксис, но и упрощают реализацию некоторых часто используемых конструкций. Подобно тому, как введение обобщенных типов много лет назад оказало значительное влияние на дальнейшее развитие Java, лямбда-выражения формируют сегодняшний облик Java. Их роль в развитии языка Java действительно весьма существенна.

Кроме того, лямбда-выражения способствовали появлению других новых возможностей Java. Об одной из них — методах интерфейсов по умолчанию — шла речь в главе 8. Эта возможность позволяет добавлять неабстрактные реализации методов в интерфейсы с помощью ключевого слова `default`. В качестве другого примера можно привести возможность использования ссылок на методы без выполнения последних. В целом введение лямбда-выражений существенно расширило возможности Java API.

Помимо непосредственной пользы, которую приносит использование лямбда-выражений, существует еще одна причина, делающая столь важной их поддержку в Java. За последние несколько лет лямбда-выражения стали важным элементом проектирования компьютерных языков. Соответствующий синтаксис включен, например, в языки C# и C++. Поэтому не последним фактором, диктовавшим необходимость включения лямбда-выражений в Java, было стремление утвердить программистов во мнении, что Java — живой, развивающийся язык, возможности которого постоянно обновляются. Данная глава посвящена обсуждению этой увлекательной темы.

Знакомство с лямбда-выражениями

Ключом к пониманию лямбда-выражений служат две конструкции: во-первых, само лямбда-выражение, а во-вторых, функциональный интерфейс. Начнем с определений каждого из этих понятий.

Лямбда-выражение — это, по сути, анонимный (т.е. неименованный) метод. Однако сам этот метод никогда не выполняется и лишь позволяет назначить реализацию кода метода, определяемого функциональным интерфейсом. Таким образом, лямбда-выражение представляет собой некую форму анонимного класса. Другой часто употребляемый эквивалентный термин в отношении лямбда-выражений — *замыкание*.

Функциональный интерфейс — это интерфейс, который содержит один и только один абстрактный метод. Обычно подобный метод определяет предполагаемое назначение интерфейса. Таким образом, функциональный интерфейс представляет, как правило, какое-то одно действие. Например, стандартный интерфейс `Runnable` является функциональным интерфейсом, поскольку в нем указан только один метод — `run()`, которым и определяется назначение интерфейса. Помимо этого, функциональный интерфейс определяет целевой тип лямбда-выражения. Здесь следует сделать одно важное замечание: лямбда-выражение может использоваться только в том контексте, в котором определен целевой тип. Стоит также отметить, что о функциональных интерфейсах иногда говорят как об интерфейсах *SAM* (Single Abstract Method — одиночный абстрактный метод).

Рассмотрим лямбда-выражения и функциональные интерфейсы более подробно.

Примечание

Функциональный интерфейс также может включать любой открытый метод, определенный в классе `Object`, например метод `equals()`, не лишаясь при этом статуса функционального интерфейса. Открытые методы класса `Object` считаются неявными членами функциональных интерфейсов, поскольку они автоматически реализуются экземплярами таких интерфейсов.

Основные сведения о лямбда-выражениях

Лямбда-выражения вводят в язык Java новый синтаксис. В них используется новый *лямбда-оператор* `->` (другое название — *оператор-стрелка*). Этот оператор разделяет лямбда-выражение на две части. В левой части указываются параметры, если того требует лямбда-выражение, а в правой — *тело лямбда-выражения*, которое описывает действия, выполняемые лямбда-выражением. В Java поддерживаются две разновидности тел лямбда-выражений. Тело одиночного лямбда-выражения состоит из одного выражения, тело блочного — из блока

кода. Мы начнем с рассмотрения лямбда-конструкций, определяющих одиночное выражение.

Прежде чем двигаться дальше, ознакомимся с несколькими конкретными примерами лямбда-выражений. Приведенное ниже лямбда-выражение является, вероятно, самым простым из тех, которые вы сможете использовать. Оно вычисляет постоянное значение:

```
() -> 98.6
```

Это лямбда-выражение не имеет параметров, поэтому список параметров пуст. Подразумевается, что возвращаемым типом является `double`. Таким образом, данное выражение эквивалентно следующему методу:

```
double myMeth() { return 98.6; }
```

Разумеется, метод, определяемый лямбда-выражением, не обладает именем.

Ниже приведен несколько более интересный пример лямбда-выражения.

```
() -> Math.random() * 100
```

Данное лямбда-выражение получает псевдослучайное значение с помощью метода `Math.random()`, умножает его на 100 и возвращает результат. Оно также не требует параметров.

В случае лямбда-выражения, имеющего параметр, он указывается в списке параметров слева от лямбда-оператора:

```
(n) -> 1.0 / n;
```

Это лямбда-выражение возвращает результат, представляющий собой обратное значение параметра `n`. Таким образом, если `n` равно `4.0`, то будет возвращено значение `0.25`. Несмотря на то что тип параметра (в данном примере параметра `n`) можно указывать явно, этого часто можно не делать, если он легко устанавливается из контекста. Как и в случае именованных методов, в лямбда-выражении можно указывать любое необходимое количество параметров.

В качестве возвращаемого типа лямбда-выражения может использоваться любой действительный тип. Например, следующее лямбда-выражение возвращает значение `true` в случае четного значения параметра `n` и `false` — в случае нечетного:

```
(n) -> (n % 2)==0;
```

Таким образом, возвращаемым типом данного лямбда-выражения является `boolean`.

Учтите также следующее: если в лямбда-выражении имеется всего один параметр, заключать его в скобки в левой части лямбда-оператора необязательно. Например, приведенная ниже форма записи лямбда-выражения является совершенно правильной.

```
n -> (n % 2)==0;
```

Чтобы избежать возможных разночтений, в данной книге списки параметров заключены в скобки во всех лямбда-выражениях, включая и те, которые имеют

только один параметр. Конечно же, в подобных случаях вы вправе придерживаться того стиля записи лямбда-выражений, который вам удобнее.

Функциональные интерфейсы

Ранее уже отмечалось, что функциональный интерфейс должен определять ровно один абстрактный метод. Прежде чем продолжить, вспомните, что не все методы интерфейса должны быть абстрактными (см. главу 8). Начиная с JDK 8 допускается, чтобы интерфейс имел один или несколько методов, используемых по умолчанию. Методы по умолчанию *не являются* абстрактными. Не являются таковыми и статические методы. Метод интерфейса является абстрактным лишь в том случае, если он не определяет какой-либо реализации. Отсюда следует, что функциональный интерфейс может включать методы по умолчанию и статические методы, но в любом случае он должен иметь один и только один абстрактный метод. Поскольку любой метод интерфейса, не определенный явно как метод по умолчанию или статический, считается абстрактным, в использовании модификатора `static` нет необходимости (хотя вы можете использовать его, если хотите).

Вот пример функционального интерфейса.

```
interface MyValue {
   double getValue();
}
```

В данном случае метод `getValue()` неявно задан как абстрактный и является единственным методом, определяемым интерфейсом `MyValue`. Следовательно, `MyValue` — функциональный интерфейс, и его функция определена как `getValue()`.

Ранее уже отмечалось, что лямбда-выражения сами по себе не выполняются. Они формируют реализацию абстрактного метода, определяемого функциональным интерфейсом, который задает свой целевой тип. Как следствие, лямбда-выражение может быть задано лишь в том контексте, в котором определен целевой тип. Один из таких контекстов создается при присвоении лямбда-выражения ссылке на функциональный интерфейс. К числу других контекстов целевого типа относятся, в частности, инициализация переменной, инструкция `return` и аргументы метода.

Обратимся к простому примеру. Сначала объявляется ссылка на функциональный интерфейс `MyValue`.

```
// Создать ссылку на экземпляр MyValue
MyValue myVal;
```

Затем этой ссылке на интерфейс назначается лямбда-выражение.

```
// Использовать лямбда-выражение в контексте присваивания
myVal = () -> 98.6;
```

Данное лямбда-выражение согласуется с объявлением `getValue()`, поскольку, подобно `getValue()`, оно не имеет параметров и возвращает результат типа

double. Вообще говоря, тип абстрактного метода, определяемого функциональным интерфейсом, должен быть совместимым с типом лямбда-выражения. Невыполнение этого условия вызовет ошибку компиляции.

Как вы, вероятно, уже догадались, при желании оба предыдущих шага можно объединить в одну инструкцию:

```
MyValue myVal = () -> 98.6;
```

где переменная myVal инициализируется лямбда-выражением.

Когда лямбда-выражение встречается в контексте целевого типа, автоматически создается экземпляр класса, который реализует функциональный интерфейс, причем лямбда-выражение определяет поведение абстрактного метода, объявленного функциональным интерфейсом. Вызов этого метода через целевой тип приводит к выполнению лямбда-выражения. Таким образом, лямбда-выражение выступает в качестве средства, позволяющего преобразовать сегмент кода в объект.

В предыдущем примере реализация метода getValue() обеспечивается лямбда-выражением. Следовательно, в результате выполнения приведенного ниже кода отобразится значение 98.6.

```
// Вызвать метод getValue(), реализованный
// ранее присвоенным лямбда-выражением
System.out.println("Постоянное значение: " + myVal.getValue());
```

Поскольку лямбда-выражение, назначенное переменной myVal, возвращает значение 98.6, это же значение будет получено и при вызове метода getValue().

Если лямбда-выражение имеет параметры, абстрактный метод функционального интерфейса также должен иметь такое же количество параметров. Рассмотрим, например, функциональный интерфейс MyParamValue, позволяющий передавать значение методу getValue().

```
interface MyParamValue {
   double getValue(double v);
}
```

Этот интерфейс можно использовать для реализации лямбда-выражения, вычисляющего обратную величину, которое приводилось в предыдущем разделе. Например:

```
MyParamValue myPval = (n) -> 1.0 / n;
```

В дальнейшем переменную myPval можно использовать, например, так.

```
System.out.println("Значение, обратное значению 4, равно " +
                   myPval.getValue(4.0));
```

Здесь метод getValue() реализован с помощью лямбда-выражения, доступного через переменную myPval, и это выражение возвращает значение, обратное значению аргумента. В данном случае методу getValue() передается значение 4.0, а возвращается значение 0.25.

В предыдущем примере есть еще нечто, представляющее интерес. Обратите внимание на то, что тип параметра n не определен. Заключение о нем делается на основании контекста. В данном случае это тип double, о чем можно судить по типу параметра метода getValue(), определяемого интерфейсом MyParamValue, каковым является тип double. Возможно также явное указание типа параметра в лямбда-выражении. Например, предыдущее выражение можно было бы записать в следующем виде:

```
(double n) -> 1.0 / n;
```

где для n явно указан тип double. Обычно необходимость в явном указании типов параметров не возникает.

Прежде чем двигаться дальше, важно обратить внимание на следующий момент: чтобы лямбда-выражение можно было использовать в контексте целевого типа, типы абстрактного метода и лямбда-выражения должны быть совместимыми. Так, если в абстрактном методе указаны два параметра типа int, то в лямбда-выражении также должны быть указаны два параметра, типы которых либо явно определены как int, либо могут неявно следовать из контекста. В общем случае типы и количество параметров лямбда-выражения должны быть совместимыми с параметрами и возвращаемым типом метода.

Применение лямбда-выражений

Теперь, когда мы достаточно подробно обсудили свойства лямбда-выражений, рассмотрим конкретные примеры их применения. В первом из них отдельные фрагменты, представленные в предыдущем разделе, собираются в завершенную программу, с которой вы сможете поэкспериментировать.

```
// Демонстрация двух простых лямбда-выражений.

// Функциональный интерфейс
interface MyValue {
   double getValue();
}

// Еще один функциональный интерфейс
interface MyParamValue {
   double getValue(double v);
}

class LambdaDemo {
   public static void main(String args[])
   {
      MyValue myVal; // объявление ссылки на интерфейс

      // Здесь лямбда-выражение — это просто константа.
      // При его назначении переменной myVal создается
      // экземпляр класса, в котором лямбда-выражение
      // реализует метод getValue() интерфейса MyValue.
      myVal = () -> 98.6;
```

Функциональные интерфейсы

Простое лямбда-выражение

```
        // Вызвать метод getValue(), предоставляемый ранее
        // назначенным лямбда-выражением
        System.out.println("Постоянное значение: " + myVal.getValue());

        // Создать параметризованное лямбда-выражение и
        // назначить его ссылке на экземпляр MyParamValue.
        // Это лямбда-выражение возвращает обратную величину
        // своего аргумента.
        MyParamValue myPval = (n) -> 1.0 / n; ◄——— Лямбда-выражение с параметром

        // Вызвать метод getValue(v) посредством ссылки myPval.
        System.out.println("Обратная величина 4 равна " +
                           myPval.getValue(4.0));
        System.out.println("Обратная величина 8 равна " +
                           myPval.getValue(8.0));

        // Лямбда-выражение должно быть совместимо с методом,
        // который определяется функциональным интерфейсом. Поэтому
        // приведенные ниже два фрагмента кода не будут работать.
//      myVal = () -> "three"; // Ошибка: тип String не совместим
                               // с типом double!
//      myPval = () -> Math.random(); // Ошибка: требуется параметр!
    }
}
```

В результате выполнения данной программы будет получен следующий результат.

```
Постоянное значение: 98.6
Обратная величина 4 равна 0.25
Обратная величина 8 равна 0.125
```

Как ранее уже упоминалось, лямбда-выражение должно быть совместимым с абстрактным методом, который вы планируете реализовать. Поэтому в приведенной выше программе последние, закомментированные строки кода недопустимы. Первая из них — из-за несовместимости типа `String` с типом `double`, т.е. возвращаемым типом метода `getValue()`, вторая — из-за того, что метод `getValue(double v)` интерфейса `MyParamValue` требует параметра, а он не предоставлен.

Важнейшим свойством функционального интерфейса является то, что его можно использовать с любым совместимым с ним лямбда-выражением. В качестве примера рассмотрим программу, определяющую функциональный интерфейс `NumericTest`, в котором объявляется абстрактный метод `test()`. Этот метод имеет два параметра типа `int`, возвращает результат типа `boolean` и предназначен для проверки передаваемых ему аргументов на предмет соответствия определенному условию. Результат проверки возвращается в виде булевого значения. В методе `main()` с помощью лямбда-выражений создаются три разных теста. В первом из них проверяется, делится ли первый аргумент на второй без остатка, во втором — меньше ли первый аргумент, чем второй, а третий

тест возвращает значение `true` в случае равенства абсолютных величин обоих аргументов. Обратите внимание на то, что каждое из лямбда-выражений, реализующих эти тесты, имеет два параметра и возвращает результат типа `boolean`. Конечно же, это является обязательным требованием и обусловлено тем, что метод `test()` имеет два параметра и возвращает результат типа `boolean`.

```
// Использование одного и того же функционального интерфейса
// с тремя различными лямбда-выражениями.

// Функциональный интерфейс имеет два параметра типа int и
// возвращает результат типа boolean.
interface NumericTest {
   boolean test(int n, int m);
}

class LambdaDemo2 {
   public static void main(String args[])
   {
      // Данное лямбда-выражение проверяет,
      // кратно ли одно число другому
      NumericTest isFactor = (n, d) -> (n % d) == 0;

      if(isFactor.test(10, 2))
         System.out.println("2 является делителем 10");
      if(!isFactor.test(10, 3))
         System.out.println("3 не является делителем 10");
      System.out.println();

      // Данное лямбда-выражение возвращает true,
      // если первый аргумент меньше второго
      NumericTest lessThan = (n, m) -> (n < m);

      if(lessThan.test(2, 10))
         System.out.println("2 меньше 10");
      if(!lessThan.test(10, 2))
         System.out.println("10 не меньше 2");
      System.out.println();

      // Данное лямбда-выражение возвращает true, если оба
      // аргумента равны по абсолютной величине
      NumericTest absEqual = (n, m) -> (n < 0 ? -n : n) ==
                                        (m < 0 ? -m : m);

      if(absEqual.test(4, -4))
         System.out.println("Абсолютные величины 4 и -4 равны");
      if(!lessThan.test(4, -5))
         System.out.println("Абсолютные величины 4 и -5 не равны");
      System.out.println();
   }
}
```

Использование одного и того же функционального интерфейса для трех разных лямбда-выражений

В результате выполнения данной программы выводится следующая информация.

```
2 является делителем 10
3 не является делителем 10

2 меньше 10
10 не меньше 2

Абсолютные величины 4 и -4 равны
Абсолютные величины 4 и -5 не равны
```

Как видите, поскольку все три лямбда-выражения совместимы с методом `test()`, все они могли быть выполнены с использованием ссылок типа `NumericTest`. В действительности в использовании трех отдельных переменных для хранения соответствующих ссылок не было необходимости, поскольку во всех трех тестах можно было использовать одну и ту же переменную. Например, можно было создать переменную `myTest` и использовать ее поочередно в каждом из тестов.

```
NumericTest myTest;

myTest = (n, d) -> (n % d) == 0;
if(myTest.test(10, 2))
   System.out.println("2 является делителем 10");
// ...
myTest = (n, m) -> (n < m);
if(myTest.test(2, 10))
   System.out.println("2 меньше 10");
//...
myTest = (n, m) -> (n < 0 ? -n : n) == (m < 0 ? -m : m);
if(myTest.test(4, -4))
   System.out.println("Абсолютные величины 4 and -4 равны");
// ...
```

Преимуществом использования различных ссылочных переменных с именами `isFactor`, `lessThan` и `absEqual`, как это было сделано в первоначальном варианте программы, является то, что при этом сразу становится ясно, на какое именно лямбда-выражение ссылается переменная.

С рассмотренной только что программой связан еще один интересный момент. Обратите внимание на то, как в ней задаются параметры лямбда-выражений. Взгляните, например, на выражение, с помощью которого проверяется кратность двух чисел:

```
(n, d) -> (n % d) == 0
```

Заметьте, что параметры `n` и `d` разделены запятой. В общем случае, когда требуется указать несколько параметров, они указываются в левой части лямбда-оператора в виде списка с использованием запятой в качестве разделителя.

В предыдущих примерах в качестве параметров и возвращаемых значений абстрактных методов, определяемых функциональными интерфейсами, использовались простые типы значений, однако на этот счет не существует ограничений. Ниже приведен пример программы, в которой объявляется функциональный интерфейс `StringTest` с абстрактным методом `test()`, имеющим два параметра типа `String` и возвращающим результат типа `boolean`. Следовательно, этот интерфейс может быть использован для тестирования некоторых условий, связанных со строками. В данной программе создается лямбда-выражение, позволяющее определить, содержится ли одна строка в другой.

```
// Функциональный интерфейс, тестирующий две строки
interface StringTest {
   boolean test(String aStr, String bStr);
}

class LambdaDemo3 {
   public static void main(String args[])
   {
      // Данное лямбда-выражение определяет,
      // является ли одна строка частью другой
      StringTest isIn = (a, b) -> a.indexOf(b) != -1;

      String str = "Это тест";

      System.out.println("Тестируемая строка: " + str);

      if(isIn.test(str, "Это"))
         System.out.println("'Это' найдено");
      else
         System.out.println("'Это' не найдено");

      if(isIn.test(str, "xyz"))
         System.out.println("'xyz' найдено");
      else
         System.out.println("'xyz' не найдено");
   }
}
```

При выполнении данной программы выводится следующая информация.

```
Тестируемая строка: Это тест
'Это' найдено
'xyz' не найдено
```

Обратите внимание на то, что для определения вхождения одной строки в другую в лямбда-выражении используется метод `indexOf()`, определенный в классе `String`. Эта методика срабатывает, поскольку тип параметров `a` и `b` автоматически определяется компилятором из контекста как `String`. Таким образом, вызов метода класса `String` для параметра `a` является вполне допустимым.

Спросим у эксперта

ВОПРОС. Ранее вы говорили о том, что в случае необходимости тип параметра лямбда-выражения можно указать в явном виде. Как быть в тех случаях, когда в лямбда-выражении имеется несколько параметров? Следует ли указывать типы всех без исключения параметров или можно предоставить компилятору возможность самостоятельно определить тип одного или нескольких из них?

ОТВЕТ. В тех случаях, когда требуется явное указание типа одного из параметров, типы всех остальных параметров в списке также должны быть указаны в явном виде. Например, ниже приведена допустимая форма записи лямбда-выражения:

```
(int n, int d) -> (n % d) == 0
```

А вот пример недопустимой формы записи лямбда-выражения:

```
(int n, d) -> (n % d) == 0
```

Недопустимым является также следующее выражение:

```
(n, int d) -> (n % d) == 0
```

Блочные лямбда-выражения

В предыдущих примерах тело каждого лямбда-выражения представляло собой одиночное выражение. В подобных случаях говорят об *одиночном (или строчном) лямбда-выражении*. Код с правой стороны лямбда-оператора в лямбда-выражениях этого типа должен состоять всего лишь из одного выражения, значение которого становится значением лямбда-оператора. Несмотря на несомненную полезность строчных лямбда-выражений, встречаются ситуации, в которых одного выражения оказывается недостаточно. Чтобы можно было справляться с такими ситуациями, в Java поддерживается другая разновидность лямбда-выражений, в которых код с правой стороны лямбда-оператора, представляющий тело выражения, может содержать несколько инструкций, записанных в виде *блока кода*. Лямбда-выражения с таким блочным телом называют *блочными*.

Появление блочных лямбда-выражений, допускающих включение нескольких инструкций в тело лямбда-оператора, позволило расширить круг возможных операций. Например, в блочных лямбда-выражениях можно объявлять переменные, использовать циклы и такие инструкции, как `if` и `switch`, создавать вложенные блоки и т.п. Создание блочного лямбда-выражение не составляет особого труда. Для этого достаточно заключить тело выражения в фигурные скобки, как это делается в случае обычных блоков инструкций.

За исключением того, что их тело состоит из нескольких инструкций, блочные лямбда-выражения используются в основном точно так же, как одиночные.

Единственное существенное отличие заключается в том, что для возврата значений в блочных лямбда-выражениях необходимо явно использовать инструкцию `return`. Это приходится делать, поскольку тело блочного лямбда-выражения содержит ряд выражений, а не одно.

Обратимся к примеру, в котором блочное лямбда-выражение используется для поиска наименьшего положительного делителя заданного целого числа. Мы будем использовать интерфейс `NumericFunc` с методом `func()`, который имеет один аргумент типа `int` и возвращает результат типа `int`.

```
// Блочное лямбда-выражение, предназначенное для нахождения
// наименьшего положительного делителя заданного целого числа.

interface NumericFunc {
   int func(int n);
}

class BlockLambdaDemo {
   public static void main(String args[])
   {

      // Данное блочное лямбда-выражение возвращает наименьший
      // положительный делитель заданного целого числа
      NumericFunc smallestF = (n) -> {
         int result = 1;

         // Получить абсолютное значение n
         n = n < 0 ? -n : n;

         for(int i=2; i <= n/i; i++)
            if((n % i) == 0) {
               result = i;
               break;
         }

         return result;
      };

      System.out.println("Наименьшим делителем 12 является " +
                         smallestF.func(12));
      System.out.println("Наименьшим делителем 11 является " +
                         smallestF.func(11));
   }
}
```

Блочное лямбда-выражение

При выполнении данной программы выводится следующая информация.

```
Наименьшим делителем 12 является 2
Наименьшим делителем 11 является 1
```

В данном блочном лямбда-выражении объявляется переменная `result`, используется цикл `for` и осуществляется возврат по инструкции `return`.

В лямбда-выражениях блочного типа это вполне допустимо. По сути, тело блочного лямбда-выражения аналогично телу метода. И еще одно замечание: когда в лямбда-выражении встречается инструкция `return`, она приводит к возврату из лямбда-выражения, но не из метода, в котором она содержится.

Обобщенные функциональные интерфейсы

Само лямбда-выражение не может определять типы параметров. Следовательно, лямбда-выражение не может быть обобщенным. (Разумеется, с учетом возможности выведения типов все лямбда-выражения могут считаться в некоторой степени обобщенными.) Однако функциональный интерфейс, связанный с лямбда-выражением, может быть обобщенным. В подобных случаях целевой тип лямбда-выражения отчасти определяется типами аргумента или аргументов, указываемыми при объявлении ссылки на функциональный интерфейс.

Попытаемся проанализировать, в чем состоит ценность обобщенных функциональных интерфейсов. Ранее мы создали два различных интерфейса: `NumericTest` и `StringTest`. Они использовались для того, чтобы определить, удовлетворяют ли два заданных значения определенным условиям. С этой целью в каждом из интерфейсов определялся свой метод `test()`, имеющий два параметра и возвращающий результат типа `boolean`. В случае интерфейса `NumericTest` тестируемыми значениями являются целые числа, а в случае интерфейса `StringTest` — строки. Таким образом, единственное, чем различаются оба метода, так это типом данных, которыми они оперируют. Такая ситуация идеальна для применения обобщенных типов. Вместо двух функциональных интерфейсов, методы которых отличаются лишь используемыми типами данных, можно объявить один обобщенный интерфейс, пригодный для использования в обоих случаях. Продемонстрируем этот подход на примере приведенной ниже программы.

```
// Использование обобщенного функционального интерфейса.

// Обобщенный функциональный интерфейс с двумя параметрами,
// который возвращает результат типа boolean
interface SomeTest<T> { ◄——— Обобщенный функциональный интерфейс
   boolean test(T n, T m);
}

class GenericFunctionalInterfaceDemo {
   public static void main(String args[])
   {
      // Данное лямбда-выражение определяет, является ли
      // одно целое число делителем другого
      SomeTest<Integer> isFactor = (n, d) -> (n % d) == 0;

      if(isFactor.test(10, 2))
         System.out.println("2 является делителем 10");
      System.out.println();
```

```
        // Данное лямбда-выражение определяет, является ли
        // одно число типа Double делителем другого
        SomeTest<Double> isFactorD = (n, d) -> (n % d) == 0;

        if(isFactorD.test(212.0, 4.0))
           System.out.println("4.0 является делителем 212.0");
        System.out.println();

        // Данное лямбда-выражение определяет, является ли
        // одна строка частью другой
        SomeTest<String> isIn = (a, b) -> a.indexOf(b) != -1;

        String str = "Обобщенный функциональный интерфейс";

        System.out.println("Тестируемая строка: " + str);

        if(isIn.test(str, "face"))
           System.out.println("'face' найдено");
        else
           System.out.println("'face' не найдено");
    }
}
```

В результате выполнения данной программы выводится следующая информация.

```
2 является делителем 10
4.0 является делителем 212.0
Тестируемая строка: Обобщенный функциональный интерфейс
'face' найдено
```

Обобщенный функциональный интерфейс SomeTest объявлен в программе следующим образом.

```
interface SomeTest<T> {
   boolean test(T n, T m);
}
```

Здесь T определяет тип обоих параметров метода test(). Это означает, что данный интерфейс совместим с произвольным лямбда-выражением, имеющим два параметра того же типа и возвращающим результат типа boolean.

Интерфейс SomeTest используется для предоставления ссылок на три типа лямбда-выражений. В первом из них используется тип Integer, во втором — тип Double, в третьем — тип String. Это позволило использовать один и тот же функциональный интерфейс для ссылок на лямбда-выражения isFactor, isFactorD и isIn. Различаются эти три случая лишь типом аргумента, передаваемого экземпляру SomeTest.

Следует отметить, что интерфейс NumericTest, рассмотренный в предыдущем разделе, также может быть переписан в виде обобщенного интерфейса, на чем построено упражнение для самопроверки, приведенное в конце главы.

Упражнение 14.1 Передача лямбда-выражения в качестве аргумента

LambdaArgumentDemo.java

Лямбда-выражения можно использовать в любом контексте, предоставляющем целевой тип. В предыдущих примерах использовались целевые контексты присваивания и инициализации. Примером контекста другого типа может служить передача лямбда-выражения методу в качестве аргумента. В действительности именно этот контекст является обычным способом использования лямбда-выражений, значительно усиливающим выразительность языка Java.

Проиллюстрируем этот процесс на примере упражнения по созданию трех строковых функций, с помощью которых выполняются следующие операции: обращение строки, обращение регистра букв в строке и замена пробелов дефисами. В упражнении эти функции реализуются в виде лямбда-выражений функционального интерфейса `StringFunc`. Каждая из функций поочередно передается методу `changeStr()` в качестве первого аргумента. Метод `changeStr()` применяет полученную строковую функцию к строке, которая задается вторым аргументом, и возвращает результат. Такой подход обеспечивает возможность применения целого ряда различных строковых функций посредством единственного метода `changeStr()`. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл `LambdaArgumentDemo.java`.
2. Добавьте в файл функциональный интерфейс `StringFunc`.

```
interface StringFunc {
   String func(String str);
}
```

   Данный интерфейс определяет метод `func()`, который имеет аргумент типа `String` и возвращает результат типа `String`. Таким образом, метод `func()` может воздействовать на строку и возвращать результат.
3. Начните создавать класс `LambdaArgumentDemo`, определив в нем метод `changeStr()`.

```
class LambdaArgumentDemo {

   // В данном методе типом первого параметра является
   // функциональный интерфейс. Следовательно, ему можно передать
   // ссылку на любой экземпляр этого интерфейса, в том числе и на
   // экземпляр, созданный посредством лямбда-выражения. С помощью
   // второго параметра задается строка, подлежащая обработке.
   static String changeStr(StringFunc sf, String s) {
      return sf.func(s);
   }
```

Как указано в комментариях, метод changeStr() имеет два параметра. Тип первого из них — StringFunc. Это означает, что методу может быть передана ссылка на любой экземпляр интерфейса StringFunc, в том числе и на экземпляр, созданный с помощью лямбда-выражения, совместимого с интерфейсом StringFunc. Строка, подлежащая обработке, передается с помощью параметра s. Возвращаемым значением является обработанная строка.

4. Начните создавать метод main().

```
public static void main(String args[])
{
   String inStr = "Лямбда-выражения расширяют Java";
   String outStr;

   System.out.println("Входная строка: " + inStr);
```

Здесь inStr — ссылка на строку, подлежащую обработке, a outStr получает измененную строку.

5. Определите лямбда-выражение, располагающее символы строки в обратном порядке, и присвойте его ссылке на экземпляр StringFunc. Заметим, что оно представляет собой еще один пример блочного лямбда-выражения.

```
// Определите лямбда-выражение, располагающее содержимое
// строки в обратном порядке, и присвойте его переменной,
// ссылающейся на экземпляр StringFunc
StringFunc reverse = (str) -> {
   String result = "";
   for(int i = str.length()-1; i >= 0; i--)
      result += str.charAt(i);

   return result;
};
```

6. Вызовите метод changeStr(), передав ему лямбда-выражение reverse и строку inStr. Присвойте результат переменной outStr и отобразите его.

```
// Передайте лямбда-выражение reverse методу changeStr()
// в качестве первого аргумента. Передайте входную строку
// в качестве второго аргумента.
outStr = changeStr(reverse, inStr);
System.out.println("Обращенная строка: " + outStr);
```

Мы можем передать лямбда-выражение reverse методу changeStr(), поскольку его первый параметр имеет тип StringFunc. Вспомните, что в результате использования лямбда-выражения создается экземпляр целевого типа, каковым в данном случае является StringFunc. Таким образом, лямбда-выражение обеспечивает эффективную передачу кода методу.

7. Завершите создание программы, добавив лямбда-выражения, заменяющие пробелы дефисами и обращающие регистр букв, как показано ниже. Заметьте, что оба лямбда-выражения непосредственно встраиваются в вызовы

метода changeStr(), тем самым избавляя нас от необходимости использовать для этого отдельные переменные типа StringFunc.

```
        // Данное лямбда-выражение заменяет пробелы дефисами.
        // Оно внедряется непосредственно в вызов
        // метода changeStr().
        outStr = changeStr((str) -> str.replace(' ', '-'), inStr);
        System.out.println("Строка с замененными пробелами: " + outStr);

        // Данное блочное лямбда-выражение обращает регистр
        // букв в строке. Оно также внедряется непосредственно
        // в вызов метода changeStr().
        outStr = changeStr((str) -> {
                              String result = "";
                              char ch;

                              for(int i = 0; i < str.length(); i++ ) {
                                 ch = str.charAt(i);
                                 if(Character.isUpperCase(ch))
                                    result += Character.toLowerCase(ch);
                                 else
                                    result += Character.toUpperCase(ch);
                              }
                              return result;
                           }, inStr);

        System.out.println("Строка с обращенным регистром букв: " +
                            outStr);
    }
}
```

Как видно из приведенного выше кода, внедрение лямбда-выражения, заменяющего пробелы на дефисы, непосредственно в вызов метода changeStr() не только уменьшает размер кода, но и облегчает его понимание. Это обусловлено простотой самого лямбда-выражения, которое содержит лишь вызов метода replace(), осуществляющего требуемую замену символов. Метод replace() определен в классе String. Используемая здесь версия этого метода принимает в качестве аргументов заменяемый и подставляемый символы и возвращает измененную строку.

В то же время непосредственное внедрение лямбда-выражения, обращающего регистр букв в строке, в вызов метода changeStr() использовано здесь исключительно в иллюстративных целях. В данном случае это породило скорее неуклюжий код, разбираться в котором довольно трудно. Обычно такие лямбда-выражения лучше передавать методу посредством использования отдельных переменных (как это было сделано при обращении порядка следования символов в строке). Но с технической точки зрения непосредственная передача лямбда-выражений методу, использованная в данном примере, также корректна.

Следует также обратить ваше внимание на то, что в лямбда-выражении, меняющем регистр букв на противоположный, используются статические методы isUpperCase(), toUpperCase() и toLowerCase(), определенные в классе Character. Вспомните, что класс Character служит оболочкой для типа char. Метод isUpperCase() возвращает значение true, если переданный ему аргумент представляет собой букву в верхнем регистре, и значение false в противном случае. Методы toUpperCase() и toLowerCase() устанавливают для букв соответственно верхний и нижний регистры и возвращают результат. Кроме этих методов, в классе Character определен ряд других методов, предназначенных для манипулирования символами и их тестирования. Более подробно об этом вы сможете узнать самостоятельно из других источников.

8. Ниже приведен полный код программы в законченном виде.

```
// Использование лямбда-выражения в качестве аргумента метода

interface StringFunc {
   String func(String str);
}

class LambdaArgumentDemo {

   // В данном методе типом первого параметра является
   // функциональный интерфейс. Это позволяет передать
   // методу ссылку на любой экземпляр данного интерфейса,
   // в том числе на экземпляр, созданный посредством
   // лямбда-выражения. С помощью второго параметра
   // задается строка, подлежащая обработке.
   static String changeStr(StringFunc sf, String s) {
      return sf.func(s);
   }

   public static void main(String args[])
   {
      String inStr = "Лямбда-выражения расширяют Java";
      String outStr;

      System.out.println("Входная строка: " + inStr);

      // Определите лямбда-выражение, располагающее содержимое
      // строки в обратном порядке, и присвойте его переменной,
      // ссылающейся на экземпляр StringFunc
      StringFunc reverse = (str) -> {
         String result = "";

         for(int i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

         return result;
      };
```

```
        // Передайте лямбда-выражение reverse методу changeStr()
        // в качестве первого аргумента. Передайте входную строку
        // в качестве второго аргумента.
        outStr = changeStr(reverse, inStr);
        System.out.println("Обращенная строка: " + outStr);

        // Данное лямбда-выражение заменяет пробелы дефисами.
        // Оно внедряется непосредственно в вызов метода changeStr().
        outStr = changeStr((str) -> str.replace(' ', '-'), inStr);
        System.out.println("Строка с замененными пробелами: " + outStr);

        // Данное блочное лямбда-выражение обращает регистр
        // букв в строке. Оно также внедряется непосредственно
        // в вызов метода changeStr().
        outStr = changeStr((str) -> {
                          String result = "";
                          char ch;

                          for(int i = 0; i < str.length(); i++ ) {
                             ch = str.charAt(i);
                             if(Character.isUpperCase(ch))
                                result += Character.toLowerCase(ch);
                             else
                                result += Character.toUpperCase(ch);
                          }
                          return result;
                       }, inStr);

        System.out.println("Строка с обращенным регистром букв: " +
                           outStr);
    }
}
```

Данная программа выводит следующую информацию.

```
Входная строка: Лямбда-выражения расширяют Java
Обращенная строка: avaJ dnapxE snoisserpxE adbmaL
Строка с замененными пробелами: Лямбда-выражения-расширяют-Java
Строка с обращенным регистром букв: lAMBDA eXPRESSIONS eXPAND jAVA
```

Спросим у эксперта

ВОПРОС. Существуют ли для лямбда-выражений другие контексты целевого типа, кроме контекстов инициализации, присваивания и передачи аргумента?

ОТВЕТ. Да, такие контексты существуют. Это операторы приведения типов, оператор ?, инициализатор массива, инструкция `return`, а также сами лямбда-выражения.

Лямбда-выражения и захват переменных

Переменные, определенные в области действия лямбда-выражения, доступны этому выражению. Например, лямбда-выражение может использовать переменную экземпляра или статическую переменную, определенную в классе, содержащем данное выражение. Лямбда-выражение также имеет доступ (как явный, так и неявный) к переменной, заданной ключевым словом this, которая ссылается на экземпляр класса, вызывающий данное выражение. Поэтому лямбда-выражение может получать и устанавливать значения переменных указанного типа и вызывать метод, определенный в содержащем данное выражение классе.

Однако, если лямбда-выражение использует локальную переменную из охватывающей области видимости, возникает особая ситуация — *захват переменной*. В подобных случаях лямбда-выражение может использовать такую переменную, но так, как если бы это была переменная типа final, значение которой не может быть изменено. Модификатор final для такой переменной можно не указывать, но если вы его укажете, то это не будет считаться ошибкой. (Параметр this, соответствующий охватывающей области видимости, ведет себя как финальная переменная, а собственного аналога переменной this лямбда-выражения не имеют.)

Важно понимать, что значение локальной переменной из охватывающей лямбда-выражение области видимости не может быть изменено выражением, поскольку это противоречило бы статусу неизменности такой переменной и сделало бы ее захват недопустимым.

Приведенная ниже программа иллюстрирует различие между переменными, которые ведут себя как финальные в лямбда-выражении, и переменными, значение которых может быть изменено.

```
// Пример захвата локальной переменной из охватывающей
// лямбда-выражение области видимости

interface MyFunc {
   int func(int n);
}

class VarCapture {
   public static void main(String args[])
   {
      // Локальная переменная, которая может быть захвачена
      int num = 10;

      MyFunc myLambda = (n) -> {
         // Такое использование переменной num корректно,
         // поскольку ее значения не изменяется
         int v = num + n;
```

```
            // Приведенная ниже инструкция некорректна,
            // поскольку она изменяет значение переменной num
//          num++;

            return v;
        };

        // Использование лямбда-выражения.
        // Эта инструкция отобразит число 18.
        System.out.println(myLambda.func(8));

        // Приведенная ниже строка породила бы ошибку, поскольку она
        // лишает num статуса финальной переменной
//      num = 9;
    }
}
```

Как отмечено в комментариях к выполняющейся части программы, переменная `num` не изменяется и может быть использована в теле `myLambda`. Поэтому в результате выполнения инструкции `println()` выводится число 18. При вызове `func()` с аргументом 8 значение `v` внутри лямбда-выражения устанавливается равным сумме `num` (значение 10) и значения, переданного параметру `n` (которое равно 8). Следовательно, `func()` возвращает число 18. Этот механизм работает, поскольку переменная `num` не изменяет свое значение после инициализации. Но если бы значение `num` было изменено — будь-то в лямбда-выражении или вне его, — переменная `num` потеряла бы свой статус неизменной (`final`) переменной. Это породило бы ошибку, препятствующую компиляции программы.

Важно подчеркнуть, что лямбда-выражение может использовать и изменять переменную экземпляра класса, в котором оно содержится. Не допускается лишь использование тех локальных переменных в области видимости, охватывающей лямбда-выражение, значения которых подвергаются изменениям.

Генерация исключений в лямбда-выражениях

Лямбда-выражения могут генерировать исключения. Однако если генерируется проверяемое исключение, то оно должно быть совместимым с исключениями, перечисленными в спецификации `throws` абстрактного метода, определяемого функциональным интерфейсом. Например, если лямбда-выражение может генерировать исключение `IOException`, то в упомянутом абстрактном методе функционального интерфейса исключение `IOException` должно быть указано в спецификации `throws`. В качестве примера рассмотрим следующую программу.

```
import java.io.*;

interface MyIOAction {
```

```
   boolean ioAction(Reader rdr) throws IOException;
}

class LambdaExceptionDemo {

   public static void main(String args[])
   {
      double[] values = { 1.0, 2.0, 3.0, 4.0 };

      // Данное блочное лямбда-выражение может генерировать
      // исключение IOException. Следовательно, это исключение
      // должно быть указано в спецификации throws метода
      // ioAction() функционального интерфейса MyIOAction.
      MyIOAction myIO = (rdr) -> { ◄── Это лямбда-выражение может генерировать исключение
         int ch = rdr.read(); // может генерировать
                              // исключение IOException

         // ...
         return true;
      };
   }
}
```

Поскольку вызов метода `read()` может сопровождаться генерацией исключения `IOException`, оно должно быть указано в спецификации `throws` метода `ioAction()` функционального интерфейса `MyIOAction`. Без этого программа не будет скомпилирована ввиду несовместимости лямбда-выражения с методом `ioAction()`. Чтобы это проверить, удалите спецификацию `throws` и попытайтесь скомпилировать программу. Вы увидите, что компилятор выведет сообщение об ошибке.

Спросим у эксперта

ВОПРОС. Можно ли использовать в лямбда-выражении параметр в виде массива?

ОТВЕТ. Да, можно. Однако в тех случаях, когда тип параметра выводится компилятором, способ указания параметра лямбда-выражения отличается от того, который обычно принят для массивов: вместо общепринятого обозначения, например `n[]`, используется простое имя переменной — `n`. Не забывайте о том, что тип параметра лямбда-выражения будет выводиться на основании целевого контекста. Таким образом, если целевой контекст требует использования массива, то в качестве типа параметра будет подразумеваться массив. Чтобы вам было легче в этом разобраться, рассмотрим следующий короткий пример.

Ниже приведен обобщенный функциональный интерфейс `MyTransform`, который может быть использован для применения некоторого преобразования к элементам массива.

```
// Функциональный интерфейс
interface MyTransform<T> {
   void transform(T[] a);
}
```

Заметьте, что параметром метода `transform()` является массив типа <T>. Создадим следующее лямбда-выражение, которое использует интерфейс `MyTransform` для преобразования элементов массива типа `Double` в их квадратные корни.

```
MyTransform<Double> sqrts = (v) -> {
   for(int i=0; i < v.length; i++) v[i] = Math.sqrt(v[i]);
};
```

Здесь типом параметра `a` метода `transform()` является тип `Double[]`, поскольку при объявлении лямбда-выражения `sqrts` для интерфейса `MyTransform` задан тип `Double`. Поэтому тип `v` в лямбда-выражении выводится как `Double[]`. Использовать для этой цели запись `v[]` было бы не только излишне, но и неправильно.

И последнее замечание: параметр для лямбда-выражения вполне мог быть объявлен как `Double[]`, поскольку это просто означало бы явное объявление типа параметра, исключающее необходимость его автоматического определения компилятором. Однако в данном случае это не дает никакого выигрыша.

Ссылки на методы

С лямбда-выражениями тесно связано одно важное средство — *ссылки на методы*. Они позволяют ссылаться на метод без его выполнения. Данное средство имеет отношение к лямбда-выражениям, поскольку для него также требуется контекст целевого типа, состоящий из совместимого функционального интерфейса. При вычислении ссылки на метод также создается экземпляр функционального интерфейса. Существует несколько разновидностей ссылок на методы. Начнем с рассмотрения ссылок на статические методы.

Ссылки на статические методы

Ссылка на статический метод создается посредством указания имени метода, которому предшествует имя класса, с использованием следующего общего синтаксиса:

имя_класса::*имя_метода*

Заметьте, что имена класса и метода разделены парой двоеточий. Символ `::` — это новый разделитель, специально добавленный в выпуске JDK 8. Данная ссылка на метод может использоваться везде, где она совместима с целевым типом.

Применение ссылок на статические методы продемонстрируем с помощью приведенной ниже программы. Сначала в программе объявляется функциональный интерфейс `IntPredicate`, имеющий метод `test()`. Этот метод имеет параметр типа `int` и возвращает результат типа `boolean`. Метод предназначен для проверки заданного целого числа на предмет удовлетворения определенным условиям. Далее в программе создается класс `MyIntPredicates`, содержащий три статических метода — `isPrime()`, `isEven()` и `isPositive()`, соответственно предназначенных для проверки того, что число является простым, четным или положительным. В классе `MethodRefDemo` создается метод `numTest()`, первым параметром которого является ссылка на `IntPredicate`. С помощью второго параметра задается целое число, подлежащее тестированию. Описанные три теста выполняются в методе `main()` посредством вызова метода `numTest()`, которому поочередно передаются ссылки на три вышеперечисленных тестовых метода.

```
// Демонстрация использования ссылок на статические методы.

// Функциональный интерфейс для числовых предикатов, которые
// воздействуют на целочисленные значения
interface IntPredicate {
   boolean test(int n);
}

// Данный класс определяет три статических метода, которые
// проверяют целое число на соответствие определенным условиям
class MyIntPredicates {
   // Статический метод, который возвращает true,
   // если заданное число простое
   static boolean isPrime(int n) {

      if(n < 2) return false;
      for(int i=2; i <= n/i; i++) {
         if((n % i) == 0)
            return false;
      }
      return true;
   }

   // Статический метод, который возвращает true,
   // если заданное число четное
   static boolean isEven(int n) {
      return (n % 2) == 0;
   }

   // Статический метод, который возвращает true,
   // если заданное число положительное
   static boolean isPositive(int n) {
      return n > 0;
   }
}
```

```
class MethodRefDemo {

   // В данном методе типом первого параметра является
   // функциональный интерфейс. Следовательно, ему можно передать
   // ссылку на любой экземпляр этого интерфейса, в том числе и на
   // экземпляр, созданный посредством ссылки на метод.
   static boolean numTest(IntPredicate p, int v) {
      return p.test(v);
   }

   public static void main(String args[])
   {
      boolean result;

      // Здесь методу numTest() передается ссылка
      // на метод isPrime()
      result = numTest(MyIntPredicates::isPrime, 17);
      if(result) System.out.println("17 - простое число");

      // Здесь методу numTest() передается ссылка
      // на метод isEven()
      result = numTest(MyIntPredicates::isEven, 12);
      if(result) System.out.println("12 - четное число");

      // Здесь методу numTest() передается ссылка
      // на метод isPositive()
      result = numTest(MyIntPredicates::isPositive, 11);
      if(result) System.out.println("11 - положительное число");
   }
}
```

Использование ссылок на статические методы

При выполнении данной программы выводится следующая информация.

```
17 - простое число
12 - четное число
11 - положительное число
```

В этой программе особый интерес представляет следующая строка:

```
result = numTest(MyIntPredicates::isPrime, 17);
```

где методу `numTest()` в качестве первого аргумента передается ссылка на статический метод `isPrime()`. Это можно было сделать, поскольку ссылка `isPrime` совместима с функциональным интерфейсом `IntPredicate`. Таким образом, вычисление выражения `MyIntPredicates::isPrime` дает ссылку на объект, метод `isPrime()` которого предоставляет реализацию метода `test()` интерфейса `IntPredicate`. Остальные два вызова метода `numTest()` работают аналогичным образом.

Ссылки на методы экземпляров

Ссылка на метод экземпляра конкретного объекта создается с применением следующего базового синтаксиса:

ссылка_на_объект::имя_метода

Как видите, приведенный синтаксис аналогичен синтаксису для ссылок на статические методы, только вместо имени класса используется объектная ссылка. Следовательно, фигурирующий здесь метод связывается с объектом, на который указывает *ссылка_на_объект*. Сказанное иллюстрирует приведенная ниже программа, в которой используются те же интерфейс `IntPredicate` и метод `test()`, что и в предыдущей программе. Однако в данном случае создается класс `MyIntNum`, в котором хранится значение типа `int` и определяется метод `isFactor()`, предназначенный для проверки того, что переданное ему число является делителем числа, хранящегося в экземпляре `MyIntNum`. Далее в методе `main()` создаются экземпляры класса `MyIntNum`. Затем для каждого из этих экземпляров поочередно вызывается метод `numTest()` с передачей ему ссылки на метод `isFactor()` соответствующего экземпляра и выполняется необходимая проверка. В каждом из этих случаев ссылка на метод привязывается к конкретному объекту.

```
// Использование ссылки на метод экземпляра.

// Функциональный интерфейс для числовых предикатов,
// которые воздействуют на целочисленные значения
interface IntPredicate {
   boolean test(int n);
}

// Данный класс хранит значение типа int и определяет метод
// isFactor(), который возвращает значение true, если его
// аргумент является делителем числа, хранящегося в классе
class MyIntNum {
   private int v;

   MyIntNum(int x) { v = x; }
   int getNum() { return v; }

   // Вернуть true, если n - делитель v
   boolean isFactor(int n) {
      return (v % n) == 0;
   }
}

class MethodRefDemo2 {

   public static void main(String args[])
   {
      boolean result;

      MyIntNum myNum = new MyIntNum(12);
      MyIntNum myNum2 = new MyIntNum(16);
```

```
        // Создать ссылку ip на метод isFactor объекта myNum
        IntPredicate ip = myNum::isFactor;

        // Использовать ссылку для вызова метода isFactor()
        // через метод test()
        result = ip.test(3);
        if(result) System.out.println("3 является делителем " +
                                      yNum.getNum());

        // Создать ссылку на метод isFactor для объекта myNum2
        // и использовать ее для вызова метода isFactor()
        // через метод test()
        ip = myNum2::isFactor;
        result = ip.test(3);
        if(!result) System.out.println("3 не является делителем " +
                                      myNum2.getNum());
    }
}
```

Ссылка на метод экземпляра

При выполнении данной программы выводится следующая информация.

```
3 является делителем 12
3 не является делителем 16
```

В этой программе особый интерес для нас представляет следующая строка:

```
IntPredicate ip = myNum::isFactor;
```

где переменной `ip` присваивается ссылка на метод `isFactor()` объекта `myNum`. Таким образом, если вызвать метод `test()`, как показано ниже

```
result = ip.test(3);
```

то он вызовет метод `isFactor()` для объекта `myNum`, т.е. того объекта, который был указан при создании ссылки на метод. Аналогичная ситуация возникает и в случае ссылки на метод `myNum2::isFactor`, если не считать того, что теперь метод `isFactor()` вызывается для объекта `myNum2`. Это подтверждается выводимой информацией.

Иногда могут возникать ситуации, требующие указания метода экземпляра, который может использоваться с любым объектом данного класса, а не только с каким-то конкретным объектом. В этом случае ссылка на метод создается с использованием следующего синтаксиса:

имя_класса::*имя_метода_экземпляра*

где вместо имени конкретного объекта указывается имя класса, даже если применяется метод экземпляра. В этой форме первый параметр функционального интерфейса соответствует типу вызывающего объекта, а второй — параметру (если таковой имеется), заданному методом. Ниже приведен соответствующий пример, представляющий собой переработанный вариант предыдущего примера. Прежде всего, интерфейс `IntPredicate` заменен интерфейсом `MyIntNumPredicate`. В этом случае первый параметр метода `test()`

имеет тип MyIntNum. Он будет использоваться для получения объекта, подлежащего обработке. Это позволяет программе создать ссылку на метод экземпляра isFactor(), который может использоваться с любым объектом MyIntNum.

```
// Использование ссылки на метод экземпляра для обращения
// к любому интерфейсу.

// Функциональный интерфейс для числовых предикатов, которые
// воздействуют на объект типа MyIntNum и целочисленное значение
interface MyIntNumPredicate {
   boolean test(MyIntNum mv, int n);
}

// Данный класс хранит значение типа int и определяет метод
// isFactor(), который возвращает true, если его аргумент
// является делителем числа, хранящегося в классе
class MyIntNum {
   private int v;

   MyIntNum(int x) { v = x; }

   int getNum() { return v; }

   // Вернуть true, если n - делитель v
   boolean isFactor(int n) {
      return (v % n) == 0;
   }
}

class MethodRefDemo3 {
   public static void main(String args[])
   {
      boolean result;

      MyIntNum myNum = new MyIntNum(12);
      MyIntNum myNum2 = new MyIntNum(16);

      // Создать ссылку inp на метод экземпляра isFactor()
      MyIntNumPredicate inp = MyIntNum::isFactor;  ◄── Ссылка на метод любого
                                                        объекта типа MyIntNum

      // Вызвать метод isFactor() для объекта myNum
      result = inp.test(myNum, 3);
      if(result)
         System.out.println("3 является делителем " +
                            myNum.getNum());

      // Вызвать метод isFactor() для объекта myNum2
      result = inp.test(myNum2, 3);
      if(!result)
         System.out.println("3 является делителем " +
                            myNum2.getNum());
   }
}
```

Спросим у эксперта

ВОПРОС. Как следует указывать ссылку на метод в случае обобщенных методов?

ОТВЕТ. Зачастую благодаря выводу типов тип аргумента обобщенного метода при получении его ссылки на метод необязательно задавать явно, но для таких случаев в Java предусмотрен синтаксис, который позволяет это сделать. Предположим, имеется следующий код.

```
interface SomeTest<T> {
   boolean test(T n, T m);
}

class MyClass {
   static <T> boolean myGenMeth(T x, T y) {
       boolean result = false;
       // ...
       return result;
   }
}
```

Тогда следующая инструкция будет корректной.

```
SomeTest<Integer> mRef = MyClass::<Integer>myGenMeth;
```

где тип аргумента для обобщенного метода `myGenMeth` задается явным образом. Обратите внимание на то, что тип аргумента указан после парного двоеточия (`::`). Существует следующее общее правило: если в качестве ссылки на метод задается обобщенный метод, то тип его аргумента указывается вслед за символами `::` перед именем метода. В тех случаях, когда задается обобщенный класс, тип аргумента указывается за именем класса и предшествует символам `::`.

Данная программа выводит следующую информацию.

```
3 является делителем 12
3 не является делителем 16
```

В этой программе особый интерес для нас представляет следующая строка:

```
MyIntNumPredicate inp = MyIntNum::isFactor;
```

В ней создается ссылка на метод экземпляра `isFactor()`, который будет работать с любым объектом типа `MyIntNum`. Например, если вызвать метод `test()` через ссылку `inp`, как показано ниже

```
result = inp.test(myNum, 3);
```

то в результате будет вызван метод `myNum.isFactor(3)`. Иными словами, объектом, для которого осуществляется вызов `myNum.isFactor(3)`, является объект `myNum`.

Примечание

Ссылка на метод может использовать ключевое слово `super` **для ссылки на версию суперкласса метода. Общие формы синтаксиса выглядят следующим образом:** `super::`*`название_метода`* **и** *`название_типа`*`.super::`*`название_метода`*. **Во второй форме параметр** *`название_типа`* **должен ссылаться на охватывающий класс или суперинтерфейс.**

Ссылки на конструкторы

Аналогично тому, как создаются ссылки на методы, можно создавать также ссылки на конструкторы. Используемый при этом синтаксис таков:

`имя_класса``::new`

Данную ссылку можно присвоить ссылке на любой функциональный интерфейс, который определяет метод, совместимый с конструктором. Рассмотрим следующий простой пример.

```
// Демонстрация использования ссылок на конструкторы.

// MyFunc - функциональный интерфейс, метод которого
// возвращает ссылку на MyClass
interface MyFunc {
   MyClass func(String s);
}

class MyClass {
   private String str;

   // Этот конструктор имеет аргумент
   MyClass(String s) { str = s; }

   // Это конструктор по умолчанию
   MyClass() { str = "";}

   // ...

   String getStr() { return str; }
}

class ConstructorRefDemo {
   public static void main(String args[])
   {
      // Создать ссылку на конструктор MyClass.
      // Поскольку метод func() интерфейса MyFunc
      // имеет аргумент, new ссылается на параметризованный
      // конструктор MyClass, а не на конструктор по умолчанию.
      MyFunc myClassCons = MyClass::new; ◄——— Ссылка на конструктор

      // Создать экземпляр MyClass посредством
      // ссылки на конструктор
      MyClass mc = myClassCons.func("Тестирование");
```

```
        // Использовать только что созданный экземпляр MyClass
        System.out.println("Строка str в mc: " + mc.getStr( ));
    }
}
```

Данная программа выводит следующую информацию:

```
Строка str в mc: Тестирование
```

Обратите внимание на то, что метод `func()` интерфейса `MyFunc` возвращает ссылку типа `MyClass` и имеет параметр типа `String`. Кроме того, класс `MyClass` определяет два конструктора. Для первого из них указан параметр типа `String`. Второй — это конструктор по умолчанию, не имеющий параметров. Рассмотрим следующую строку:

```
MyClass mc = myClassCons.func("Тестирование");
```

По сути, `myClassCons` предлагает другой способ выполнения вызова `MyClass(String s)`.

Если бы вы захотели использовать запись `MyClass::new` для вызова конструктора по умолчанию класса `MyClass`, то вам понадобился бы функциональный интерфейс, который определяет метод, не имеющий параметров. Например, если вы определите функциональный интерфейс `MyFunc2`, как показано ниже

```
interface MyFunc2 {
   MyClass func();
}
```

то с помощью следующей инструкции сможете присвоить переменной `myClassCons` ссылку на конструктор по умолчанию (т.е. не имеющий параметров) класса `MyClass`:

```
MyFunc2 myClassCons = MyClass::new;
```

В общем случае при использовании ссылок вида `::new` будет вызываться конструктор, параметры которого соответствуют параметрам, указанным в функциональном интерфейсе.

Спросим у эксперта

ВОПРОС. Могу ли я объявить ссылку на конструктор, создающий массив?

ОТВЕТ. Это возможно. Для создания ссылки на конструктор массива используется следующий синтаксис:

тип`[]::new`

где *тип* — тип создаваемого объекта. Пусть, например, имеются класс `MyClass` из предыдущего примера и следующий интерфейс `MyClassArrayCreator`.

```
interface MyClassArrayCreator {
   MyClass[] func(int n);
}
```

Тогда приведенный ниже фрагмент кода создает массив объектов `MyClass` и присваивает каждому его элементу начальное значение.

```
MyClassArrayCreator mcArrayCons = MyClass[]::new;
MyClass[] a = mcArrayCons.func(3);
for(int i=0; i < 3; i++)
   a[i] = new MyClass(i);
```

Здесь в результате вызова `func(3)` создается массив, состоящий из трех элементов. Этот пример можно обобщить. Любой функциональный интерфейс, предназначенный для создания массива, должен содержать метод, который имеет единственный параметр типа `int` и возвращает ссылку на массив заданного размера.

Вам также будет интересно узнать, что ничто не мешает создать обобщенный функциональный интерфейс, пригодный для использования с другими типами классов.

```
interface MyArrayCreator<T> {
   T[] func(int n);
}
```

Используя эту возможность, можно, например, создать массив, состоящий из пяти объектов типа `Thread`.

```
MyArrayCreator <Thread> mcArrayCons = Thread[]::new;
Thread[] thrds = mcArrayCons.func(5);
```

И последнее: в случае создания ссылки на конструктор для обобщенного класса вы сможете указать тип параметра обычным способом — вслед за именем класса. Например, если имеется класс, объявленный, как показано ниже:

```
MyGenClass<T> { // ...
```

то следующий код создаст ссылку на конструктор с типом аргумента `Integer`:

```
MyGenClass<Integer>::new;
```

С учетом вывода типов компилятором явное указание типа аргумента требуется не всегда, но в случае необходимости это может быть сделано.

Предопределенные функциональные интерфейсы

До этого момента во всех примерах данной главы использовались создаваемые специально для них функциональные интерфейсы, что облегчило объяснение фундаментальных понятий, лежащих в основе функциональных интерфейсов и лямбда-выражений. Однако во многих случаях можно не определять собственные функциональные интерфейсы, а использовать предопределенные интерфейсы, содержащиеся в новом пакете `java.util.function`, добавленном в JDK 8. Некоторые из них представлены в приведенной ниже таблице.

Интерфейс	Назначение
UnaryOperator<T>	Применение унарной операции к объекту типа T и возврат результата, также имеющего тип T. Название метода этого интерфейса — apply()
BinaryOperator<T>	Применение операции к двум к объектам типа T и возврат результата, также имеющего тип T. Название метода этого интерфейса — apply()
Consumer<T>	Применение операции, ассоциированной с объектом типа T. Название метода этого интерфейса — accept()
Supplier<T>	Возврат объекта типа T. Название метода этого интерфейса — get()
Function<T, R>	Применение операции к объекту типа T и возврат результата в виде объекта типа R. Название метода этого интерфейса — apply()
Predicate<T>	Определение того, удовлетворяет ли объект типа T некоторому ограничению. Возвращает результат типа boolean, указывающий на исход проверки. Название метода этого интерфейса — test()

Ниже приведен пример программы, в которой используется интерфейс Predicate. В данном случае он служит функциональным интерфейсом для лямбда-выражения, которое проверяет четность числа. Вот как выглядит объявление абстрактного метода test() интерфейса Predicate:

```
boolean test(T val)
```

Этот метод должен возвращать значение true, если val удовлетворяет некоторому ограничению или условию. В данном случае он возвращает значение true, если val — четное число.

```
// Использование встроенного функционального интерфейса Predicate.

// Импортировать интерфейс Predicate
import java.util.function.Predicate;

class UsePredicateInterface {
   public static void main(String args[])
   {

      // Данное лямбда-выражение использует интерфейс
      // Predicate<Integer> для определения того, четно ли
      // заданное число
      Predicate<Integer> isEven = (n) -> (n % 2) == 0;  ← Использование встроенного интерфейса Predicate

      if(isEven.test(4)) System.out.println("4 - четное число");

      if(!isEven.test(5)) System.out.println("5 - нечетное число");
   }
}
```

Данная программа выводит следующую информацию:

```
4 - четное число
5 - нечетное число
```

Спросим у эксперта

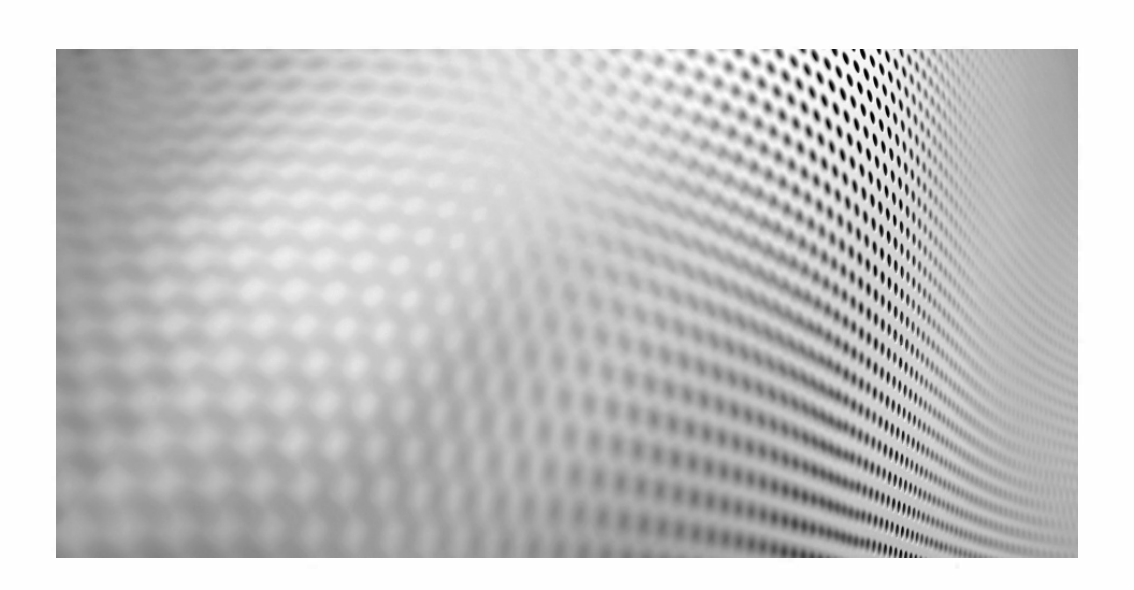
ВОПРОС. В начале главы вы упомянули о том, что в результате включения лямбда-выражений стандартная библиотека Java обогатилась новыми встроенными возможностями? Можете привести какой-либо пример?

ОТВЕТ. К числу наиболее важных улучшений Java, добавленных в JDK 8, относится новый пакет `java.util.stream`. В этом пакете определено несколько классов для работы с потоками, наиболее общим из которых является класс `Stream`. Потоки, входящие в пакет `java.util.stream`, служат средством передачи данных. Таким образом, поток представляет собой последовательность объектов. Кроме того, поток поддерживает многочисленные операции, с помощью которых можно создавать конвейеры, выполняющие ряд последовательных действий над данными. Часто для представления этих действий используются лямбда-выражения. Например, потоковый API позволяет конструировать цепочки действий, напоминающие запросы к базе данных, для выполнения которых можно было бы использовать инструкции SQL. Более того, во многих случаях эти действия могут выполняться параллельно, что повышает производительность, особенно в случае больших объемов данных. Проще говоря, потоковый API предоставляет в ваше распоряжение мощные и вместе с тем простые в использовании средства обработки данных. Следует сделать важное замечание: несмотря на некоторое сходство между новым потоковым API и потоками ввода-вывода, рассмотренными в главе 10, они не эквивалентны.

Вопросы и упражнения для самопроверки

1. Что такое лямбда-оператор?
2. Что такое функциональный интерфейс?
3. Какая связь существует между функциональными интерфейсами и лямбда-выражениями?
4. Назовите два общих типа лямбда-выражений.
5. Составьте лямбда-выражение, которое возвращает значение `true`, если число принадлежит к диапазону чисел 10–20, включая граничные значения.
6. Создайте функциональный интерфейс, способный поддерживать лямбда-выражение, предложенное в п. 5. Назовите интерфейс `MyTest`, а его абстрактный метод — `testing()`.

7. Создайте блочное лямбда-выражение для вычисления факториала целого числа. Продемонстрируйте его использование. В качестве функционального интерфейса используйте интерфейс NumericFunc, который рассматривался в этой главе.
8. Создайте обобщенный функциональный интерфейс MyFunc<T>. Назовите его абстрактный метод func(). Метод func() должен иметь параметр типа T и возвращать ссылку типа T. (Таким образом, интерфейс MyFunc должен представлять собой обобщенную версию интерфейса NumericFunc, который рассматривался в этой главе.) Продемонстрируйте его использование, переработав свое решение для п. 7 таким образом, чтобы вместо интерфейса NumericFunc в нем использовался интерфейс MyFunc<T>.
9. Используя программу, созданную в упражнении 14.1, создайте лямбда-выражение, которое удаляет все пробелы из заданной строки и возвращает результат. Продемонстрируйте работу этого метода, передав его методу changeStr().
10. Можно ли использовать в лямбда-выражении локальную переменную? Если это так, то какие при этом существуют ограничения?
11. Справедливо ли следующее утверждение: "Если лямбда-выражение может генерировать проверяемое исключение, то абстрактный метод функционального интерфейса должен содержать инструкцию throws, в которой указано данное исключение"?
12. Что такое ссылка на метод?
13. При вычислении ссылки на метод создается экземпляр ____________ ____________, предоставляемого целевым контекстом.
14. Предположим, имеется класс MyClass, содержащий статический метод myStaticMethod(). Продемонстрируйте, каким образом можно указать ссылку на метод myStaticMethod().
15. Предположим, имеется класс MyClass, содержащий объектный метод myInstMethod(), и относящийся к этому классу объект mcObj. Продемонстрируйте, как можно создать ссылку на метод myInstMethod(), ассоциированный с объектом mcObj.
16. В программе MetodRefDemo2 добавьте в класс MyIntNum новый метод hasCommonFactor(). Этот метод должен возвращать true, если его аргумент типа int и значение, которое хранится в вызывающем объекте MyIntNum, имеют по крайней мере один общий делитель. Продемонстрируйте работу метода hasCommonFactor(), используя ссылку на метод.
17. Как определяется ссылка на конструктор?
18. В каком пакете Java содержатся определения встроенных функциональных интерфейсов?



Глава 15

Модули

В этой главе...

- Знакомство с модулями
- Ключевые слова, относящиеся к модулям Java
- Объявление модуля с помощью ключевого слова `module`
- Использование инструкций `requires` и `exports`
- Назначение файла `module-info.java`
- Компиляция и запуск модульных программ с помощью `javac` и `java`
- Назначение модуля `java.base`
- Принципы поддержки унаследованного кода, применявшегося до появления модулей
- Экспорт пакета для указанного модуля
- Использование скрытой читаемости
- Применение служб в модуле

С появлением JDK 9 в Java было добавлено новое средство, которое получило название *модуль*. Благодаря модулям можно описывать связи и зависимости в коде, образующем приложение. Можно также контролировать доступность частей модуля для других модулей. Кроме того, модули позволяют создавать более надежные и лучше масштабируемые программы.

Как правило, модули широко применяются при разработке крупных приложений, поскольку позволяют уменьшить издержки, связанные с управлением большими программными системами. Но даже маленькие программы лучше создавать с помощью модулей, поскольку библиотека Java API организована по модульному принципу. Благодаря этому можно указывать, какие конкретно части API будут применяться в программе. В результате уменьшается объем ресурсов, требуемых при выполнении программ, что существенно облегчает разработку приложений для таких небольших устройств, как, например, компоненты интернета вещей (Internet of Things — IoT).

Поддержка модулей осуществляется как с помощью элементов языка, включая новые ключевые слова, так путем оптимизации `javac`, `java` и других средств JDK. Также появились новые инструменты и форматы файлов. Как следствие, поддержка модулей была добавлена в JDK и в подсистему времени выполнения JDK 9. Одним словом, благодаря модулям язык Java перешел на качественно новый уровень. В этой главе будут рассмотрены основные возможности модулей и способы их применения в приложениях Java.

Знакомство с модулями

В широком смысле под *модулем* понимается совокупность пакетов и ресурсов, доступных по имени модуля. *Объявление модуля* определяет имя модуля и связи между модулем (и его пакетами) и другими модулями. Объявления модулей — это инструкции в исходном файле Java, включающие несколько новых ключевых слов, которые имеют отношение к модулям. Эти ключевые слова приведены в следующей таблице.

exports	module	open	opens
provides	requires	to	transitive
uses	with		

Отметим, что данные слова распознаются как *ключевые слова* только в контексте объявления модулей. В иных случаях эти слова трактуются как идентификаторы. Например, ключевое слово `module` может использоваться в качестве имени параметра, хотя это и не рекомендуется.

Объявление модуля содержится в файле `module-info.java`. Таким образом, модуль определяется в исходном файле Java, который называется *дескриптором модуля*. Затем этот файл компилируется с помощью `javac` в файл класса. Файл `module-info.java` включает лишь определение модуля и не является файлом общего назначения.

Объявление модуля начинается с ключевого слова `module`, как показано в следующем примере кода:

```
module имя_модуля {
   // определение модуля
}
```

Название модуля определяется с помощью идентификатора *имя_модуля*, представляющего собой корректный идентификатор Java или последовательность идентификаторов, разделенных пробелами. Определение модуля указывается в фигурных скобках.

Спросим у эксперта

ВОПРОС. Почему новые ключевые слова, связанные с модулями, например `module` и `requires`, считаются ключевыми словами только в контексте объявления модуля?

ОТВЕТ. Благодаря использованию этих ключевых слов исключительно в области объявления модуля предотвращаются возможные проблемы с ранее разработанным кодом, в котором эти ключевые слова могут применяться в качестве идентификаторов. Например, рассмотрим ситуацию, когда программа,

созданная с помощью JDK, предшествующего версии JDK 9, применяет ключевое слово `requires` в качестве имени переменной. После переноса этой программы в JDK 9 признание `requires` в качестве ключевого слова за пределами области объявления модулей может привести к появлению ошибки компиляции. Если же `requires` признается в качестве ключевого слова исключительно в области объявления модулей, применение этого слова в любой другой области программы не приведет к появлению ошибки. Конечно же, это касается и других ключевых слов, связанных с модулями. Будучи контекстно-зависимыми, ключевые слова, связанные с модулями, формально называются *ограниченными ключевыми словами*.

Обычно в области объявления модуля содержится несколько инструкций, которые определяют его характеристики. Также определение модуля может быть пустым (в таком случае оно включает просто имя модуля).

Простой пример модуля

Возможности модуля основаны на двух ключевых свойствах. Первое из них заключается в указании на необходимость использования другого модуля. Иначе говоря, один модуль заявляет о своей зависимости от другого модуля. Отношения зависимости определяются с помощью инструкции `requires`. По умолчанию наличие требуемого модуля проверяется на этапе компиляции и на этапе выполнения. Второе ключевое свойство заключается в возможности модуля управлять доступностью своих пакетов для другого модуля, что достигается с помощью инструкции `exports`. Открытые и защищенные типы внутри пакета доступны для других модулей только при явном экспортировании. В этом разделе рассматривается пример использования обоих ключевых свойств модуля.

Данный пример демонстрирует процесс создания модульного приложения, в котором применяются простые математические функции. Мы намеренно приводим пример очень небольшого приложения, чтобы продемонстрировать общие принципы и процедуры, необходимые для создания, компиляции и выполнения модульного кода. Более того, этот же подход можно задействовать для решения реальных задач при работе с крупными приложениями. Настоятельно рекомендуем самостоятельно пошагово проработать этот пример на своем компьютере.

Примечание

В этой главе рассматривается процесс создания, компиляции и выполнения модульного кода с помощью инструментов командной строки. Данная методика имеет два основных преимущества. Во-первых, она доступна всем разработчикам Java, так как не требует интегрированной среды разработки (IDE). Во-вторых, эта методика четко демонстрирует общие принципы подсистемы модулей, включая способы применения каталогов.

Чтобы понять, как это работает, следует вручную создать несколько каталогов и убедиться в корректности размещения каждого файла. Несомненно, в ходе создания реальных модульных приложений вы обнаружите, что модульно-зависимую интегрированную среду разработки проще применять, так как она позволяет автоматизировать основные процессы. Тем не менее советуем детально изучить принципы использования модулей с помощью инструментов командной строки, чтобы сформировать более глубокое понимание данной темы.

Приложение определяет два модуля. Первый из них называется `appstart`. В нем содержится пакет `appstart.mymodappdemo`, в котором определяется точка входа приложения в классе `MyModAppDemo`, т.е. класс `MyModAppDemo` содержит метод приложения `main()`. Второй модуль называется `appfuncs`. Он содержит пакет `appfuncs.simplefuncs` с классом `SimpleMathFuncs`. Этот класс определяет три статических метода, которые реализуют простые математические функции. Все приложение находится в дереве каталогов, которое начинается с каталога `mymodapp`.

Прежде чем продолжить, остановимся на некоторых правилах именования модулей. Обратите внимание на то, что в приведенных ниже примерах имя модуля (например, `appfuncs`) становится префиксом названия пакета (например, `appfuncs.simplefuncs`), который содержится в модуле. Это правило *не* является обязательным, но оно позволяет идентифицировать модуль, к которому относится пакет. В процессе изучения модулей лучше применять в основном короткие простые имена, как, например, в этой главе. Но разрешается использовать любые удобные названия. При создании модулей, предназначенных для распространения, выбирайте имена с особой тщательностью, так как они должны быть уникальными. Рекомендуем воспользоваться методом обратного доменного имени. Суть его состоит в том, что в качестве префикса модуля выбирается обратное имя домена, которому “принадлежит” проект. Например, в проекте, связанном с сайтом `herbschildt.com`, в качестве префикса имени модуля можно использовать `com.herbschildt`. (То же самое касается и названий пакетов.) Поскольку правила именования модулей в Java появились относительно недавно, они могут со временем измениться. Рекомендации по наименованию модулей изложены в справочных пособиях по Java.

А теперь приступим к практике. Выполните следующие действия, чтобы создать необходимые каталоги исходного кода.

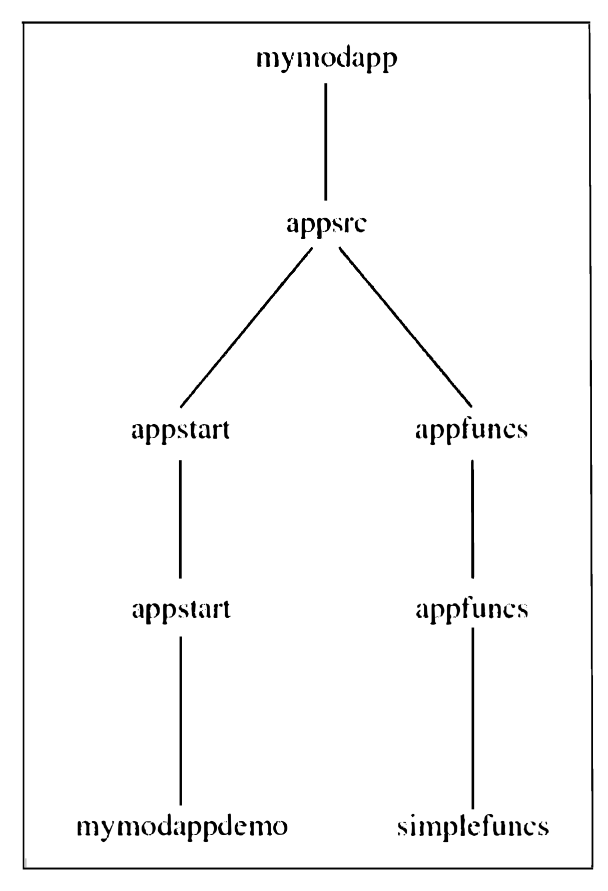
1. Создайте каталог верхнего уровня приложения под названием `mymodapp`.
2. В каталоге `mymodapp` создайте подкаталог `appsrc`, который будет каталогом верхнего уровня для исходного кода приложения.
3. В каталоге `appsrc` создайте подкаталог `appstart`, а в нем — еще один подкаталог `appstart`. В этом каталоге создайте подкаталог `mymodappdemo`. Полученное в результате дерево будет иметь следующий вид:

```
appsrc\appstart\appstart\mymodappdemo
```

4. В каталоге appsrc создайте подкаталог appfuncs, а в нем — еще один подкаталог, который также называется appfuncs. В последнем создайте подкаталог simplefuncs. Созданное в результате дерево будет иметь следующий вид:

```
appsrc\appfuncs\appfuncs\simplefuncs
```

На приведенном ниже рисунке показано только что созданное дерево каталогов.



Завершив настройку каталогов, можно приступать к созданию исходных файлов приложения.

В этом примере используются четырех исходных файла. Два из них определяют приложение. Ниже представлено содержимое первого файла под названием SimpleMathFuncs.java. Обратите внимание на то, что класс SimpleMathFuncs включен в пакет appfuncs.simplefuncs.

```
// Несколько простых математических функций

package appfuncs.simplefuncs;  ◄—— Обратите внимание на объявление пакета
```

```
public class SimpleMathFuncs {

   // Определить, является ли a множителем b
   public static boolean isFactor(int a, int b) {
      if((b%a) == 0) return true;
      return false;
   }

   // Вернуть наименьший положительный
   // общий множитель для a и b
   public static int lcf(int a, int b) {
      // Вычисление множителя на основе положительных значений
      a = Math.abs(a);
      b = Math.abs(b);

      int min = a < b ? a : b;

      for(int i = 2; i <= min/2; i++) {
         if(isFactor(i, a) && isFactor(i, b))
            return i;
      }

      return 1;
   }

   // Вернуть наибольший общий множитель для a и b
   public static int gcf(int a, int b) {
      // Вычисление множителя на основе положительных значений
      a = Math.abs(a);
      b = Math.abs(b);

      int min = a < b ? a : b;

      for(int i = min/2; i >= 2; i--) {
         if(isFactor(i, a) && isFactor(i, b))
            return i;
      }
      return 1;
   }
}
```

Модуль `SimpleMathFuncs` определяет три простые статические математические функции. Первая из них, `isFactor()`, возвращает значение `true`, если `a` является множителем `b`. Метод `lcf()` возвращает наименьший общий множитель для `a` и `b`, а метод `gcf()`, наоборот, возвращает наибольший общий множитель для `a` и `b`. В обоих случаях возвращается 1, если не найдено ни одного общего множителя. Данный файл помещается в следующий каталог:

```
appsrc\appfuncs\appfuncs\simplefuncs
```

Это каталог пакета `appfuncs.simplefuncs`.

Ниже приведено содержимое второго исходного файла MyModAppDemo.java. В нем используются методы из класса SimpleMathFuncs. Обратите внимание на то, что этот файл включен в пакет appstart.mymodappdemo и импортирует класс SimpleMathFuncs, который нужен ему для работы.

```
// Демонстрация простого модульного приложения
package appstart.mymodappdemo;  ◄——— Обратите внимание
                                      на объявление пакета
                                      и инструкцию import
import appfuncs.simplefuncs.SimpleMathFuncs;  ◄———

public class MyModAppDemo {
   public static void main(String[] args) {

      if(SimpleMathFuncs.isFactor(2, 10))
         System.out.println("2 является множителем 10");

      System.out.println("Наименьшим общим множителем для 35 и 105
                          будет " + SimpleMathFuncs.lcf(35, 105));
      System.out.println("Наибольшим общим множителем для 35 и 105
                          будет " + SimpleMathFuncs.gcf(35, 105));
   }
}
```

Этот файл находится в каталоге

```
appsrc\appstart\appstart\mymodappdemo
```

который соответствует пакету appstart.mymodappdemo.

После этого в каждый модуль следует добавить файлы module-info.java, в которых содержатся определения модулей. Сначала добавляется файл, который определяет модуль appfuncs.

```
// Определение модуля функций
module appfuncs {  ◄——— Определение модуля appfuncs
   // Экспорт пакета appfuncs.simplefuncs
   exports appfuncs.simplefuncs;
}
```

Обратите внимание на то, что модуль appfuncs экспортирует пакет appfuncs.simplefuncs, делая его доступным для других модулей. Этот файл должен находиться в следующем каталоге:

```
appsrc\appfuncs
```

Таким образом, файл будет находиться в каталоге модуля appfuncs, который находится уровнем выше каталогов пакетов.

И напоследок добавьте файл module-info.java для модуля appstart, как показано ниже. Заметьте, что модуль appstart требует использования модуля appfuncs.

```
// Определение главного модуля приложения
module appstart {  ◄——— Определение модуля appstart
   // Требуется модуль appfuncs
   requires appfuncs;
}
```

Файл помещается в каталог модуля:

```
appsrc\appstart
```

Прежде чем перейти к подробному изучению инструкций `requires`, `exports` и `module`, скомпилируйте и выполните этот пример. Проверьте правильность создания каталогов и корректность размещения файлов в них.

Компиляция и выполнение первого примера модуля

Начиная с JDK 9 в компилятор `javac` была добавлена поддержка модулей. Таким образом, аналогично всем приложениям Java модульные программы компилируются с помощью `javac`. Этот несложный процесс отличается лишь указанием конкретного ***пути к модулю***, благодаря которому компилятор получает информацию о местах размещения скомпилированных файлов. Выполняя данный пример, следите за корректным выполнением команд `javac` из каталога `mymodapp`. Не забывайте о том, что в приложении модуля каталог `mymodapp` является каталогом верхнего уровня.

Начнем с компиляции файла `SimpleMathFuncs.java` путем выполнения следующей команды.

```
javac -d appmodules\appfuncs
   appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java
```

Эта команда должна выполняться из каталога `mymodapp`. Опция `-d` сообщает компилятору `javac` о том, куда следует помещать выходной файл `.class`. Во всех примерах этой главы на вершине дерева каталогов скомпилированного кода будет каталог `appmodules`. При необходимости эта команда автоматически создает выходные каталоги для пакета `appfuncs.simplefuncs` в каталоге `appmodules\appfuncs`.

Далее с помощью команды `javac` компилируется файл `module-info.java` для модуля `appfuncs`:

```
javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java
```

В результате файл `module-info.class` помещается в каталог `appmodules\appfuncs`.

Данная пошаговая инструкция приводится здесь лишь в качестве примера. Гораздо проще скомпилировать файл модуля `module-info.java` и его исходные файлы в одной командной строке. Ниже показан результат комбинирования двух предыдущих команд `javac` в одну.

```
javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java
   appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java
```

В этом случае каждый скомпилированный файл помещается в соответствующий каталог модуля или пакета.

Теперь скомпилируем файлы `module-info.java` и `MyModAppDemo.java` модуля `appstart` с помощью следующей команды.

```
javac --module-path appmodules -d appmodules\appstart
   appsrc\appstart\module-info.java
   appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

Обратите внимание на опцию `--module-path`. С ее помощью определяется путь к модулю, что позволяет компилятору идентифицировать пользовательские модули, необходимые для файла `module-info.java`. В данном случае идентифицируется модуль `appfuncs`, который необходим для модуля `appstart`. Важно отметить, что каталог вывода определяется как `appmodules\appstart`. Это значит, что файл `module-info.class` будет находиться в каталоге модуля `appmodules\appstart`, а файл `MyModAppDemo.java` — в каталоге пакета `appmodules\appstart\appstart\mymodappdemo`.

По завершении процесса компиляции запустить приложение можно с помощью следующей команды:

```
java --module-path appmodules -m
   appstart/appstart.mymodappdemo.MyModAppDemo
```

В данном случае опция `--module-path` определяет путь к модулям приложения. Как уже упоминалось, `appmodules` — это каталог, находящийся на вершине дерева каталогов скомпилированного кода. Опция `-m` определяет класс, который содержит точку входа приложения. В данном случае это класс, содержащий метод `main()`. Выполнение программы даст следующий результат.

```
2 является множителем 10
Наименьшим общим множителем для 35 и 105 будет 5
Наибольшим общим множителем для 35 и 105 будет 7
```

Подробное знакомство с инструкциями `requires` и `exports`

Предыдущий пример модуля основывался на двух основных свойствах подсистемы модулей: возможности определять зависимость и возможности удовлетворять зависимость. Эти возможности реализуются путем использования инструкций `requires` и `exports` в объявлении модуля. Рассмотрим каждую из них в отдельности.

Инструкция `requires`, используемая в примере, имеет следующий синтаксис:

```
requires имя_модуля;
```

где параметр *имя_модуля* определяет имя модуля, который требуется для модуля, включающего инструкцию `requires`. Это значит, что требуемый модуль необходим для компиляции текущего модуля. Говоря языком модулей, текущий модуль ***читает*** модуль, который задан с помощью инструкции `requires`. По сути, инструкция `requires` позволяет убедиться в наличии доступа приложения к необходимым модулям.

Общий синтаксис инструкции `exports`, используемой в примере, таков:

```
exports имя_пакета;
```

Здесь параметр *имя_пакета* определяет имя пакета, экспортируемого модулем, в котором встречается эта инструкция. Модуль, который экспортирует пакет, делает все содержащиеся в пакете открытые и защищенные типы доступными для других модулей. Более того, открытые и защищенные члены этих типов также становятся доступными. В то же время, если пакет, находящийся в модуле, не экспортируется, значит, он является закрытым для этого модуля, включая все его открытые типы. Например, даже если класс в пакете объявлен как `public`, но не экспортируется явно с помощью инструкции `exports`, он будет недоступен для других модулей. Следует понимать, что открытые и защищенные типы, относящиеся к пакету, независимо от того, экспортируется этот пакет или нет, всегда доступны в модуле, содержащем пакет. Инструкция `exports` делает их доступными для других модулей. Таким образом, любой пакет, который не является экспортируемым, может применяться только в данном модуле.

Важно усвоить, что инструкции `requires` и `exports` используются в паре. Если один модуль зависит от другого, то он должен обозначить эту зависимость с помощью инструкции `requires`. А модуль, от которого зависит другой модуль, должен открыто экспортировать (т.е. сделать доступными) все пакеты, необходимые для зависимого модуля. При отсутствии соответствующей спецификации с одной из сторон отношения зависимый модуль не будет скомпилирован. Например, в нашем примере класс `MyModAppDemo` использует функции класса `SimpleMathFuncs`. В результате объявление модуля `appstart` содержит инструкцию `requires`, которая указывает на модуль `appfuncs`. А объявление модуля `appfuncs` экспортирует пакет `appfuncs.simplefuncs`, благодаря чему становятся доступными открытые типы в классе `SimpleMathFuncs`. Теперь обе стороны отношения зависимости определены, поэтому приложение можно скомпилировать и выполнить. В противном случае компиляция завершится неудачей. (Выполнив тестовое задание №10 в конце главы, вы узнаете, что может произойти в случае отсутствия инструкции `exports`.).

Стоит подчеркнуть, что инструкции `requires` и `exports` входят только в состав объявления `module`. Более того, само по себе объявление `module` должно содержаться в файле `module-info.java`.

Платформенные модули и пакет `java.base`

Как уже упоминалось в начале главы, начиная с версии JDK 9 пакеты Java API включаются в модули. Фактически модульность API является одним из ключевых преимуществ, обеспечиваемым модульной подсистемой. В силу своей особой роли модули API называют ***платформенными***, и в их именах используется префикс `java`, например `java.base`, `java.desktop`, `java.xml` и т.п. Благодаря модульности API можно развертывать приложения, включающие только самые необходимые пакеты, а не всю среду выполнения Java (JRE). Учитывая большие размеры JRE, модульность оказывается весьма полезным усовершенствованием.

Поскольку все пакеты Java API теперь упакованы в модули, возникает вопрос: каким образом метод `main()` в классе `MyModAppDemo` в предыдущем примере вызывал метод `System.out.println()` без указания инструкции `requires` для модуля, содержащего класс `System`? Очевидно, что приложение не скомпилируется и не запустится на выполнение без класса `System`. Аналогичный вопрос возникает и при использовании класса `Math` в классе `SimpleMathFuncs`. Чтобы ответить на этот вопрос, следует рассмотреть модуль `java.base`.

Среди платформенных модулей самым важным является модуль `java.base`, который включает и экспортирует основные для Java пакеты `java.lang`, `java.io`, `java.util` и многие другие. В силу своей значимости модуль `java.base` ***автоматически доступен*** для других модулей. Более того, все другие модули автоматически требуют модуль `java.base`. Соответственно, инструкцию `requires java.base` можно не добавлять в объявление модуля (правда, явное указание `java.base` тоже не является ошибкой). Аналогично тому, как пакет `java.lang` автоматически доступен для всех программ без применения инструкции `import`, модуль `java.base` автоматически доступен для всех модульных приложений без явного запроса с их стороны.

В силу того, что модуль `java.base` содержит пакет `java.lang`, а последний включает класс `System`, класс `MyModAppDemo` в предыдущем примере может автоматически использовать метод `System.out.println()` без явного обращения к инструкции `requires`. То же самое касается использования класса `Math` в классе `SimpleMathFuncs`, поскольку первый также содержится в пакете `java.lang`. В процессе создания собственных модульных приложений вы быстро поймете, что многие из необходимых классов API находятся в пакетах, включенных в модуль `java.base`. Таким образом, автоматическое добавление модуля `java.base` упрощает создание модульного кода благодаря автоматической доступности ключевых пакетов Java.

И последнее, на что хотелось бы обратить внимание. Начиная с JDK 9 в документации Java API указывается имя модуля, содержащего тот или иной пакет. И если это модуль `java.base`, значит, вы сможете непосредственно использовать содержимое данного пакета. В противном случае объявление вашего модуля должно включать инструкцию `requires` для требуемого модуля.

Спросим у эксперта

ВОПРОС. В JDK 8 можно было использовать такое средство, как *компактные профили*. Входят ли они в состав модулей?

ОТВЕТ. Компактные профили — это средство, которое в определенных ситуациях позволяет задавать подмножество библиотеки API. Они не являются частью модульной подсистемы. Более того, модульная подсистема, представленная в JDK 9, полностью заменяет их.

Унаследованный код и безымянный модуль

При рассмотрении первого примера модульной программы у вас мог возникнуть следующий вопрос. Если JDK 9 теперь поддерживает модули и платформенные пакеты API также содержатся в модулях, то почему другие приложения, приведенные в предыдущих главах, компилируются и выполняются без ошибок даже без использования модулей? Иначе говоря, Java существует уже более 20 лет, и за все это время большая часть созданного кода не использовала модули. Как в таком случае можно компилировать, выполнять и поддерживать унаследованный код совместно с JDK 9 или более поздними версиями компилятора? Согласно философии Java "напиши один раз, выполняй где угодно", ответ на этот вопрос предполагает поддержку обратной совместимости. И действительно, в Java применяются оригинальные способы обеспечения обратной совместимости, позволяющие использовать ранее созданный код.

Поддержка унаследованного кода обеспечивается с помощью двух ключевых средств. Первое из них — *безымянный модуль*. Код, который не является частью именованного модуля, автоматически становится частью безымянного. Безымянный модуль имеет два важных свойства: во-первых, все пакеты в нем автоматически экспортируются, во-вторых, он может иметь доступ к любым другим модулям. Таким образом, если приложение не использует модули, то все модули Java API становятся автоматически доступными благодаря безымянным модулям.

Второй способ поддержки унаследованного кода заключается в автоматическом использовании пути класса вместо пути модуля. При компиляции приложения, в котором не применяются модули, используется механизм пути класса, как это было в первых выпусках Java. В результате программа компилируется и выполняется таким же образом, как и до появления JDK 9.

В рассмотренных до сих пор примерах кода объявление модулей не использовалось благодаря безымянному модулю и автоматическому использованию пути класса. Модули выполняются корректно, независимо от того, как они компилируются — компилятором JDK 9 или более ранним, например JDK 8. Таким образом, даже несмотря на серьезное расширение инструментария Java, модули поддерживают совместимость с унаследованным кодом, обеспечивая плавный переход к модульной системе. Самое главное, такой подход позволяет не использовать модули там, где они не нужны.

Прежде чем продолжить, упомянем еще один важный момент. В приводимых в книге примерах программ, да и в демонстрационных программах вообще, применять модули не нужно. Это только усложняет работу и никак не влияет на эффективность. Более того, в процессе изучения Java можно в принципе не прибегать к использованию модулей, когда речь идет о написании простого кода. В начале главы упоминалось о том, что модули применяются главным образом при создании крупных коммерческих приложений. Именно поэтому примеры использования модулей ограничиваются рамками этой главы. Подобный подход к использованию модулей позволяет компилировать и выполнять

примеры в среде, предшествующей JDK 9, что важно для пользователей, работающих с более ранними версиями Java. Примеры, рассматриваемые в этой книге (за исключением данной главы), применимы как для модульных, так и безмодульных версий JDK.

Выполнение экспорта для определенного модуля

Базовая форма инструкции `exports` открывает доступ к пакету любым модулям. Зачастую это именно то, что требуется. Тем не менее в некоторых случаях лучше сделать пакет доступным не для *всех* модулей, а для конкретного *набора* модулей. Например, предположим, что разработчик библиотеки намерен экспортировать пакет поддержки для определенных модулей вместо того, чтобы делать его доступным для общего использования. Для этого следует добавить в инструкцию `exports` предложение `to`.

В инструкции `exports` предложение `to` определяет перечень модулей, получающих доступ к экспортируемому пакету. Говоря языком модулей, предложение `to` создает так называемый *квалифицированный экспорт*.

Ниже представлен синтаксис инструкции `exports`, которая включает предложение `to`.

```
exports имя_пакета to имена_модулей;
```

Здесь *имена_модулей* — это разделенный запятыми список модулей, которым открывается доступ из экспортирующего модуля.

Чтобы использовать предложение `to`, измените файл `module-info.java` для модуля `appfuncs`, как показано ниже.

```
// Определение модуля, которое использует предложение to
module appfuncs {
   // Экспорт пакета appfuncs.simplefuncs в модуль appstart
   exports appfuncs.simplefuncs to appstart; ◄——— Квалифицированный экспорт
}
```

Теперь пакет `simplefuncs` экспортируется только в модуль `appstart` и ни в какой другой. После этого можно заново скомпилировать приложение с помощью следующей команды:

```
javac -d appmodules --module-source-path appsrc
   appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

После завершения компиляции можно выполнить приложение, как было показано выше.

В этом примере используется другая особенность модулей, предлагаемая в JDK 9. Обратите внимание на предыдущую команду `javac`. Во-первых, она содержит опцию `--module-source-path`. Заданный в ней путь определяет вершину дерева исходных кодов модулей. Опция `--module-source-path` вызывает автоматическую компиляцию файлов в дереве, находящихся ниже указанного каталога, которым в данном примере является `appsrc`. Эта опция должна использоваться совместно с опцией `-d`, чтобы гарантировать

сохранение скомпилированных модулей в надлежащих подкаталогах каталога appmodules. Такая форма записи команды javac называется ***многомодульным режимом***, поскольку позволяет компилировать несколько модулей одновременно. Многомодульный режим компиляции особенно уместен в данном примере, поскольку предложение to задает конкретный модуль, и соответствующий модуль должен иметь доступ к экспортируемому пакету. Таким образом, во избежание появления предупреждений и ошибок в ходе компиляции необходимы оба модуля: appstart и appfuncs. Многомодульный режим устраняет любые проблемы, поскольку оба модуля компилируются одновременно.

Многомодульный режим команды javac также обеспечивает другое преимущество, которое заключается в том, что происходит автоматическое обнаружение и компиляция всех исходных файлов приложения с созданием необходимых каталогов вывода. Благодаря указанным преимуществам этот режим будет использоваться в последующих примерах.

Примечание

Квалифицированный экспорт представляет собой специальное средство. Чаще всего модули обеспечивают неквалифицированный экспорт пакета либо вообще не экспортируют его, вследствие чего пакет остается недоступным. По этой причине квалифицированный экспорт рассматривается здесь главным образом ради полноты изложения. Кроме того, квалифицированный экспорт сам по себе не предотвращает ненадлежащее использование экспортируемого пакета вредоносным кодом в модуле, маскирующимся под целевой модуль. В данной книге не рассматриваются меры безопасности, необходимые для предотвращения таких случаев. Эти вопросы подробно изложены в документации Oracle.

Использование инструкции requires transitive

Рассмотрим ситуацию, когда три модуля, A, B и C, включают следующие зависимости:

A требует B;

B требует C.

Понятно, что если A зависит от B и B зависит от C, то A имеет косвенную зависимость от C. Если A напрямую не использует содержимое C, то в файле module-info модуль A будет требовать модуль B, а B будет экспортировать пакеты, требуемые для A, в своем файле module-info.

```
// Файл module-info для модуля A
module A {
   requires B;
}
```

```
// Файл module-info для модуля B
module B {
   exports некоторый_пакет;
   requires C;

}
```

В этом примере *некоторый_пакет* является заполнителем имени пакета, который экспортируется модулем B и используется в модуле A. Данная схема работает, пока модулю A не потребуется применить что-нибудь, определенное в модуле C. В таком случае есть два варианта решения проблемы.

Первый вариант заключается в добавлении инструкции `requires C` в файл объявления модуля A.

```
// Файл module-info модуля A обновлен для явного
// запроса модуля C
module A {
   requires B;
   requires C; // также требуется C
}
```

Это решение несомненно работоспособно, но если модуль B будет использоваться множеством модулей, то придется добавлять строку `requires C` ко всем определениям модулей, работающих с B. Это утомительный процесс, к тому же чреватый появлением ошибок. К счастью, существует более эффективное решение. Можно создать ***неявную зависимость*** от C, которую еще называют ***неявным чтением***.

Для создания неявной зависимости следует добавить ключевое слово `transitive` в инструкцию `requires`. В данном примере следует изменить файл `module-info` для модуля B.

```
// Файл module-info для модуля B
module B {
   exports некоторый_пакет;
   requires transitive C;
}
```

В данном примере зависимость от модуля C становится транзитивной. В результате такой замены любой модуль, зависящий от B, будет автоматически зависеть от C. Таким образом, A автоматически получает доступ к C.

Существует интересный нюанс, касающийся синтаксиса инструкции `requires`. Если сразу после ключевого слова `transitive` стоит разделитель (например, точка с запятой), то оно интерпретируется не как ключевое слово, а как идентификатор (например, имя модуля).

Упражнение 15.1

Эксперименты с ключевыми словами requires transitive

```
MyModAppDemo.java
SimpleFuncs.java
SupportFuncs.java
module-info.java
```

А теперь попробуем поэкспериментировать с инструкцией requires transitive путем переработки предыдущего примера модульного приложения. Удалим метод isFactor() из класса SimpleMathFuncs, относящегося к пакету appfuncs.simplefuncs, и переместим его в новый класс, модуль и пакет. Новый класс будет называться SupportFuncs, модуль — appsupport, а пакет — appsupport.supportfuncs. Модуль appfuncs добавит зависимость от модуля appsupport с помощью инструкции requires transitive, что позволит обоим модулям, appfuncs и appstart, получить доступ к модулю appsupport. В модуле appstart не придется указывать для этого отдельную инструкцию requires. Подобное возможно благодаря тому, что модуль appstart получает транзитивный доступ за счет инструкции requires transitive в модуле appfuncs.

1. Сначала создайте исходные каталоги для поддержки нового модуля appsupport. Для этого в каталоге appsrc создайте подкаталог appsupport. Это каталог модуля для функций поддержки. В каталоге appsupport создайте каталог пакета, добавив подкаталог appsupport, а затем — подкаталог supportfuncs. В результате дерево каталогов appsupport будет выглядеть следующим образом:

   ```
   appsrc\appsupport\appsupport\supportfuncs
   ```

2. В исходный каталог модуля appsupport, которым является appsrc\appsupport, добавьте файл module-info.java.

   ```
   // Определение модуля appsupport
   module appsupport {
      exports appsupport.supportfuncs;
   }
   ```

3. В каталог пакета appsupport.supportfuncs добавьте файл с именем SupportFuncs.java.

   ```
   // Функции поддержки

   package appsupport.supportfuncs;

   public class SupportFuncs {
      // Определить, является ли a множителем b
      public static boolean isFactor(int a, int b) {
         if((b%a) == 0) return true;
         return false;
      }
   }
   ```

   Обратите внимание на то, что метод isFactor() теперь находится не в пакете SimpleMathFuncs, а в пакете SupportFuncs.

4. **Удалите метод `isFactor()` из пакета `SimpleMathFuncs`. Файл `SimpleMathFuncs.java` теперь имеет следующий вид.**

```
// Несколько простых математических функций.
// На этот раз метод isFactor() удален.

package appfuncs.simplefuncs;
import appsupport.supportfuncs.SupportFuncs;

public class SimpleMathFuncs {

   // Вернуть наименьший положительный
   // общий множитель для a и b
   public static int lcf(int a, int b) {
      // Вычисление множителя на основе положительных значений
      a = Math.abs(a);
      b = Math.abs(b);

      int min = a < b ? a : b;

      for(int i = 2; i <= min/2; i++) {
         if(SupportFuncs.isFactor(i, a) &&
            SupportFuncs.isFactor(i, b))
            return i;
      }

      return 1;
   }

   // Вернуть наибольший общий множитель для a и b
   public static int gcf(int a, int b) {
      // Вычисление множителя на основе положительных значений
      a = Math.abs(a);
      b = Math.abs(b);

      int min = a < b ? a : b;

      for(int i = min/2; i >= 2; i--) {
         if(SupportFuncs.isFactor(i, a) &&
            SupportFuncs.isFactor(i, b))
            return i;
      }

      return 1;
   }
}
```

Обратите внимание, что на этот раз импортируется класс `SupportFuncs`, а вызовы метода `isFactor()` осуществляются с помощью имени класса `SupportFuncs`.

5. **Измените файл `module-info.java` модуля `appfuncs` таким образом, чтобы в инструкции `requires` модуль `appsupport` определялся как `transitive`.**

```
// Определение модуля appfuncs
module appfuncs {
   // Экспорт пакета appfuncs.simplefuncs
   exports appfuncs.simplefuncs;

   // Требуется модуль appsupport, который будет транзитивным
   requires transitive appsupport;
}
```

6. **Поскольку модуль `appfuncs` помечает требуемый модуль `appsupport` как `transitive`, в файле `module-info.java` для модуля `appstart` нет необходимости указывать зависимость от модуля `appsupport`. Эта зависимость теперь становится неявной. Таким образом, в файл `module-info.java` для модуля `appstart` не нужно вносить никаких изменений.**

7. **Обновите файл `MyModAppDemo.java`, чтобы отразить все изменения. Теперь он должен импортировать класс `SupportFuncs` и указывать его при вызове метода `isFactor()`.**

```
// Обновлен для использования класса SupportFuncs
package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;
import appsupport.supportfuncs.SupportFuncs;

public class MyModAppDemo {
   public static void main(String[] args) {

      // Теперь на метод isFactor() ссылаются через класс
      // SupportFuncs, а не SimpleMathFuncs
      if(SupportFuncs.isFactor(2, 10))
         System.out.println("2 является множителем 10");

      System.out.println("Наименьшим общим множителем для 35 и 105
                          будет " + SimpleMathFuncs.lcf(35, 105));
      System.out.println("Наибольшим общим множителем для 35 и 105
                          будет " + SimpleMathFuncs.gcf(35, 105));
   }
}
```

8. **Повторно скомпилируйте программу с помощью следующей команды многомодульной компиляции:**

```
javac -d appmodules --module-source-path appsrc
   appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

Как уже упоминалось выше, многомодульная компиляция автоматически создает параллельные подкаталоги модулей в каталоге `appmodules`.

9. Выполните приложение с помощью такой команды:

```
java --module-path appmodules -m
   appstart/appstart.mymodappdemo.MyModAppDemo
```

Получим такой же результат, как и раньше.

10. В качестве эксперимента удалите спецификатор `transitive` из файла `module-info.java` для модуля `appfuncs` и выполните повторную компиляцию. Вы получите ошибку, которая возникнет вследствие недоступности модуля `appsupport` из модуля `appstart`.

11. Проведем еще один эксперимент. В файле `module-info` для модуля `appsupport` попробуйте экспортировать пакет `appsupport.supportfuncs` только в модуль `appfuncs` путем использования квалифицированного экспорта:

```
exports appsupport.supportfuncs to appfuncs;
```

Затем попробуйте повторно скомпилировать приложение. Вы увидите, что программа не будет компилироваться, поскольку теперь функция поддержки `isFactor()` недоступна в классе `MyModAppDemo`, который находится в модуле `appstart`. Как упоминалось ранее, квалифицированный экспорт ограничивает доступ к пакету только тем модулям, которые заданы с помощью опции `to`.

Использование служб

В программировании обычно стараются разделять описания того, *что* нужно делать и *как* это делать. Как упоминалось в главе 8, в языке Java это достигается с помощью интерфейсов. Интерфейс определяет, *что* делается, а реализация класса — *как* это делается. Можно пойти еще дальше и сделать так, чтобы реализация класса предоставлялась с помощью программного кода, находящегося вне программы, в ***подключаемом модуле***. Благодаря применению подобного подхода можно расширить, усовершенствовать или изменить возможности приложения, просто выбрав другой подключаемый модуль. При этом основа приложения не меняется. В Java поддерживается архитектура подключаемых приложений благодаря использованию ***служб*** и ***провайдеров служб***. Поскольку они играют важную роль, особенно в больших коммерческих приложениях, они поддерживаются подсистемой модулей Java.

Приложения, использующие службы и провайдеры служб, как правило, имеют более сложную структуру. Именно поэтому программисты довольно редко прибегают к использованию модулей, ориентированных на службы. Но в связи с тем что поддержка служб широко внедрена в подсистему модулей, важно иметь общее представление о принципах работы служб. В этом разделе рассматривается простой пример использования основных возможностей служб.

Общие сведения о службах и провайдерах служб

В Java *службой* называется программная единица, функциональность которой определяется интерфейсом или абстрактным классом. Таким образом, служба описывает общее поведение программы. Конкретная реализация службы предоставляется *провайдером*. Другими словами, служба определяет характер того или иного действия, а провайдер реализует само действие.

Как уже упоминалось, службы используются для поддержки архитектуры подключаемых модулей. Например, служба может обеспечивать перевод с одного языка на другой. В таких случаях служба реализует функции перевода в целом, тогда как провайдер обеспечивает определенное направление перевода, например с немецкого на английский или с французского на китайский. Поскольку все провайдеры служб имеют одинаковый интерфейс, для перевода на разные языки могут использоваться разные переводчики, не требующие внесения изменений в ядро приложения. Для изменения языка и направления перевода можно просто поменять провайдера службы.

Провайдеры служб поддерживаются классом `ServiceLoader`, который представляет собой обобщенный класс, находящийся в пакете `java.util`. Этот класс объявляется следующим образом:

```
class ServiceLoader<S>
```

где `S` определяет тип службы. Провайдеры служб загружаются с помощью метода `load()`, который представлен в нескольких формах. В нашем примере используется следующая форма:

```
public static <S> ServiceLoader<S> load(Class <S> тип_службы)
```

В данном примере *тип_службы* определяет объект `Class` для необходимого типа службы. В главе 13 уже упоминалось о том, что в объекте `Class` находится информация о классе. Для создания экземпляра объекта `Class` предусмотрено множество способов. В нашем случае будет использоваться *литерал класса*, который обычно принимает следующую форму:

```
имя_класса.class
```

В результате вызова метода `load()` возвращается экземпляр класса `ServiceLoader` для приложения. Этот объект поддерживает итерации и может использоваться в цикле `for-each`. Чтобы найти требуемый провайдер службы, достаточно выполнить поиск в цикле.

Ключевые слова, используемые при работе со службами

Модули поддерживают службы с помощью ключевых `provides`, `uses` и `with`. Если нужно указать, что модуль реализует службу, используйте инструкцию `provides`. Для указания службы, которая требуется модулю, предназначена инструкция `uses`. Если же нужно указать конкретный тип провайдера службы, воспользуйтесь ключевым словом `with`. Благодаря совместному

использованию этих ключевых слов можно задать модуль, предоставляющий службу, и модуль, которому требуется эта служба, а также задать конкретную реализацию службы. Более того, модульная подсистема обеспечивает доступность и обнаружение службы и провайдеров служб.

Инструкция `provides` имеет следующий синтаксис:

```
provides тип_службы with типы_реализации;
```

где *тип_службы* определяет конкретный тип службы. Обычно это интерфейс, хотя используются также абстрактные классы. Перечень типов реализации, разделенных запятыми, задается с помощью параметра *типы_реализации*. Таким образом, чтобы предоставить службу, модуль указывает как имя службы, так и ее реализацию.

Инструкция `uses` имеет следующий синтаксис:

```
uses тип_службы;
```

где *тип_службы* задает тип требуемой службы.

Пример использования модульной службы

Для демонстрации использования службы добавим ее в пример модульного приложения. Для простоты начнем с первой версии приложения, которая рассматривалась в начале главы. Добавим к этой версии два новых модуля. Первый из них — `userfuncs`. Он определяет интерфейсы с поддержкой функций, выполняющих бинарные операции, в которых каждый аргумент имеет тип `int`, а результат также принимает тип `int`. Второй модуль называется `userfuncsimp` и содержит конкретные реализации интерфейсов.

Для начала создадим необходимые исходные каталоги.

1. В каталоге `appsrc` создайте подкаталоги `userfuncs` и `userfuncsimp`.
2. В каталог `userfuncs` добавьте подкаталог с таким же именем (`userfuncs`), а в него — подкаталог `binaryfuncs`. В результате получится дерево, на вершине которого находится каталог `appsrc`:

   ```
   appsrc\userfuncs\userfuncs\binaryfuncs
   ```

3. В каталог `userfuncsimp` добавьте подкаталог с таким же именем (`userfuncsimp`), а в него — подкаталог `binaryfuncsimp`. В результате получится дерево, на вершине которого находится подкаталог `appsrc`:

   ```
   appsrc\userfuncsimp\userfuncsimp\binaryfuncsimp
   ```

Этот пример расширяет исходную версию приложения путем предоставления поддержки для функций, которые не встроены в приложение. Напомним, что класс `SimpleMathFuncs` содержит три встроенные функции: `isFactor()`, `lcf()` и `gcf()`. В него можно добавить дополнительные функции, но для этого потребуется модифицировать и повторно скомпилировать приложение. В то же время благодаря использованию служб можно подключать новые функции в

процессе выполнения программы, не прибегая к ее изменению. Именно это мы и осуществим в данном примере. В нашем случае служба предоставляет функции, которые имеют два аргумента типа `int` и возвращают результат типа `int`. Следует отметить, что при наличии дополнительных интерфейсов могут поддерживаться и другие типы функций, но поддержка целочисленных бинарных функций является достаточной для наших целей и обеспечивает компактность исходного кода.

Интерфейсы служб

Мы создадим два интерфейса служб, один из которых определяет тип действия, а второй — тип провайдера этого действия. Оба интерфейса находятся в каталоге `binaryfuncs` и входят в пакет `userfuncs.binaryfuncs`. Первый интерфейс, `BinaryFunc`, объявляет спецификацию бинарной функции.

```
// Данный интерфейс определяет функцию, которая имеет
// два аргумента типа int и возвращает результат типа int.
// Это может быть любая бинарная операция с двумя аргументами
// типа int, которая возвращает результат типа int.

package userfuncs.binaryfuncs;

public interface BinaryFunc {
   // Определение имени функции
   public String getName();

   // Это выполняемая функция. Она будет
   // обеспечена конкретными реализациями.
   public int func(int a, int b);
}
```

Интерфейс `BinaryFunc` объявляет структуру объекта, который может реализовывать бинарную функцию, определяемую методом `func()`. Имя функции можно узнать с помощью метода `getName()`. Оно будет использоваться для определения разновидности реализуемой функции. Сам интерфейс реализуется классом, предоставляющим бинарную функцию.

Второй интерфейс, `BinFuncProvider`, объявляет структуру провайдера службы. Код этого интерфейса приведен ниже.

```
// Этот интерфейс определяет структуру провайдера службы,
// который возвращает экземпляры BinaryFunc
package userfuncs.binaryfuncs;

import userfuncs.binaryfuncs.BinaryFunc;

public interface BinFuncProvider {

   // Получение экземпляра BinaryFunc
   public BinaryFunc get();
}
```

Интерфейс BinFuncProvider объявляет только один метод, get(), который используется для получения экземпляра BinaryFunc. Этот интерфейс должен реализовываться классом, который хочет предоставлять экземпляры интерфейса BinaryFunc.

Классы реализации

В данном примере поддерживаются две конкретные реализации интерфейса BinaryFunc. Первая из них, AbsPlus, возвращает сумму абсолютных значений аргументов. Вторая, AbsMinus, возвращает результат вычитания абсолютного значения второго аргумента из абсолютного значения первого аргумента. Эти реализации предоставляются классами AbsPlusProvider и AbsMinusProvider. Исходный код этих классов хранится в каталоге binaryfuncsimp и включен в пакет userfuncsimp.binaryfuncsimp.

Код класса AbsPlus приведен ниже.

```
// Класс AbsPlus обеспечивает конкретную реализацию интерфейса
// BinaryFunc. Он возвращает результат abs(a) + abs(b).

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.BinaryFunc;

public class AbsPlus implements BinaryFunc {

   // Возвращает имя функции
   public String getName() {
      return "absPlus";
   }

   // Реализация функции AbsPlus
   public int func(int a, int b) { return Math.abs(a) + Math.abs(b); }
}
```

Реализация метода func(), применяемого для суммирования абсолютных значений

Код класса AbsPlus реализует метод func() таким образом, что он возвращает результат сложения абсолютных величин a и b. Обратите внимание на то, что метод getName() возвращает строку "absPlus". Это идентификатор функции.

Ниже представлен код класса AbsMinus.

```
// Класс AbsMinus обеспечивает конкретную реализацию интерфейса
// BinaryFunc. Он возвращает результат abs(a) - abs(b).

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.BinaryFunc;

public class AbsMinus implements BinaryFunc {

   // Возвращает имя функции
```

```
    public String getName() {
        return "absMinus";
    }

    // Реализация функции AbsMinus
    public int func(int a, int b) { return Math.abs(a) - Math.abs(b); }
}
```

Реализация метода func(), применяемого для вычитания абсолютных значений

В данном примере функция func() возвращает разницу между абсолютными значениями a и b, а метод getName() возвращает строку "absMinus".

Для получения экземпляра класса AbsPlus используется провайдер AbsPlusProvider, который реализует интерфейс BinFuncProvider.

```
// Провайдер для функции AbsPlus

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.*;

public class AbsPlusProvider implements BinFuncProvider {

    // Предоставляет объект AbsPlus
    public BinaryFunc get() { return new AbsPlus(); }
}
```

Возвращает объект AbsPlus

Метод get() возвращает новый объект AbsPlus(). Это очень простой провайдер, но нужно понимать, что зачастую провайдеры служб оказываются гораздо более сложными.

Провайдер для функции AbsMinus называется AbsMinusProvider и имеет следующий вид.

```
// Провайдер для функции AbsMinus

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.*;

public class AbsMinusProvider implements BinFuncProvider {

    // Предоставляет объект AbsMinus
    public BinaryFunc get() { return new AbsMinus(); }
}
```

Возвращает объект AbsMinus

Его метод get() возвращает объект AbsMinus.

Файлы определения модуля

Создадим два файла определения модуля. Первый из них предназначен для модуля userfuncs.

```
module userfuncs {
    exports userfuncs.binaryfuncs;
}
```

Этот код должен содержаться в файле `module-info.java`, который находится в каталоге модуля `userfuncs`. Отметим, что он экспортирует пакет `userfuncs.binaryfuncs`, который определяет интерфейсы `BinaryFunc` и `BinFuncProvider`.

Второй файл `module-info.java` определяет модуль, который содержит реализации интерфейсов. Он находится в каталоге модуля `userfuncsimp`.

```
module userfuncsimp {
   requires userfuncs;

   provides userfuncs.binaryfuncs.BinFuncProvider with
      userfuncsimp.binaryfuncsimp.AbsPlusProvider,
      userfuncsimp.binaryfuncsimp.AbsMinusProvider;
}
```

Этому модулю требуется модуль `userfuncs`, в котором содержатся интерфейсы `BinaryFunc` и `BinFuncProvider`, необходимые для реализаций. Модуль предоставляет реализации интерфейса `BinFuncProvider` вместе с классами `AbsPlusProvider` и `AbsMinusProvider`.

Использование провайдеров служб в приложении MyModAppDemo

Чтобы продемонстрировать использование служб, мы расширим метод `main()` в программе `MyModAppDemo`, задействовав в нем функции `AbsPlus` и `AbsMinus`. Они будут загружаться на этапе выполнения с помощью метода `ServiceLoader.load()`. Вот как выглядит обновленный код приложения.

```
// Модульное приложение, демонстрирующее использование
// служб и провайдеров служб

package appstart.mymodappdemo;

import java.util.ServiceLoader;

import appfuncs.simplefuncs.SimpleMathFuncs;
import userfuncs.binaryfuncs.*;

public class MyModAppDemo {
   public static void main(String[] args) {

      // Сначала используются встроенные службы, как и прежде
      if(SimpleMathFuncs.isFactor(2, 10))
         System.out.println("2 является множителем 10");

      System.out.println("Наименьшим общим множителем для 35 и 105
                          будет " + SimpleMathFuncs.lcf(35, 105));
      System.out.println("Наибольшим общим множителем для 35 и 105
                          будет " + SimpleMathFuncs.gcf(35, 105));

      // Теперь используем основанные на службах
      // пользовательские операции.
```

```
      // Получение загрузчика службы для бинарных функций
      ServiceLoader<BinFuncProvider> ldr =  ← Загрузка служб
         ServiceLoader.load(BinFuncProvider.class);

      BinaryFunc binOp = null;

      // Поиск провайдера для функции absPlus и получение функции
      for(BinFuncProvider bfp : ldr) {
         if(bfp.get().getName().equals("absPlus")) {  ← Поиск провайдера для операции сложения абсолютных значений
            binOp = bfp.get();
            break;
         }
      }

      if(binOp != null)
         System.out.println("Результат выполнения функции
                            absPlus: " + binOp.func(12, -4));
      else
         System.out.println("Функция absPlus не найдена");

      binOp = null;

      // Теперь ищем провайдера для функции absMinus и получаем функцию
      for(BinFuncProvider bfp : ldr) {
         if(bfp.get().getName().equals("absMinus")) {  ← Поиск провайдера для операции вычитания абсолютных значений
            binOp = bfp.get();
            break;
         }
      }

      if(binOp != null)
         System.out.println("Результат выполнения функции
                            absMinus: " + binOp.func(12, -4));
      else
         System.out.println("Функция absMinus не найдена");
   }
}
```

Рассмотрим подробнее, как загружается и выполняется служба в приведенном выше коде. Сначала с помощью следующей инструкции создается загрузчик для служб типа `BinFuncProvider`:

```
ServiceLoader<BinFuncProvider> ldr =
   ServiceLoader.load(BinFuncProvider.class);
```

Обратите внимание на то, что параметром типа для `ServiceLoader` является `BinFuncProvider`. Этот же тип указан в вызове метода `load()`. В результате происходит обнаружение провайдеров, реализующих этот интерфейс, т.е. после завершения инструкции классы, реализующие интерфейс `BinFuncProvider` в модуле, будут доступны через объект загрузчика `ldr`. В нашем случае будут доступны классы `AbsPlusProvider` и `AbsMinusProvider`.

Далее объявляется ссылка типа `BinaryFunc` под названием `binOp`, которая инициализируется значением `null`. Она будет использоваться для ссылки

на реализацию с конкретным типом бинарной функции. В следующем цикле в объекте `ldr` ищется реализация с именем `"absPlus"`.

```
// Поиск провайдера для функции absPlus и получение функции
for(BinFuncProvider bfp : ldr) {
   if(bfp.get().getName().equals("absPlus")) {
      binOp = bfp.get();
      break;
   }
}
```

В цикле `for-each` последовательно перебирается содержимое загрузчика `ldr`. В цикле проверяется имя функции, предоставляемой провайдером. Если оно совпадает с `"absPlus"`, то ссылка на объект функции присваивается переменной `binOp` путем вызова метода провайдера `get()`.

В результате, если функция обнаружена, как в данном примере, то она выполняется с помощью следующей инструкции.

```
if(binOp != null)
   System.out.println("Результат выполнения функции
                        absPlus: " + binOp.func(12, -4));
```

В данном случае, поскольку переменная `binOp` ссылается на экземпляр класса `AbsPlus`, метод `func()` выполняет сложение абсолютных величин. Такая же последовательность действий используется для поиска и выполнения функции `AbsMinus`.

В силу того что программа `MyModAppDemo` теперь использует интерфейс `BinFuncProvider`, файл определения модуля должен включать соответствующую инструкцию `uses`. Напомним, что класс `MyModAppDemo` содержится в модуле `appstart`. Поэтому следует внести изменения в файл `module-info.java` для модуля `appstart`, как показано ниже.

```
// Определение модуля для главного модуля приложения.
// Теперь используется интерфейс BinFuncProvider.
module appstart {
   // Требуются модули appfuncs и userfuncs
   requires appfuncs;
   requires userfuncs;

   // Модуль appstart теперь использует интерфейс BinFuncProvider
   uses userfuncs.binaryfuncs.BinFuncProvider;
}
```

Компиляция и запуск службы модуля

Выполнив все предыдущие действия, можно скомпилировать и запустить пример с помощью следующих команд.

```
javac -d appmodules --module-source-path appsrc
   appsrc\userfuncsimp\module-info.java
   appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java

java --module-path appmodules -m
   appstart/appstart.mymodappdemo.MyModAppDemo
```

Результат выполнения программы выглядит так.

```
2 является множителем 10
Наименьшим общим множителем для 35 и 105 будет 5
Наибольшим общим множителем для 35 и 105 будет 7
Результат выполнения функции absPlus: 16
Результат выполнения функции absMinus: 8
```

Этот пример демонстрирует успешный поиск и выполнение бинарных функций. Важно подчеркнуть, что при отсутствии инструкции `provides` в модуле `userfuncsimp` или инструкции `uses` в модуле `appstart` приложение не будет выполнено.

Дополнительные возможности модулей

Прежде чем подводить итоги, рассмотрим еще три инструкции, связанные с модулями: `open module`, `opens` и `requires static`. Каждая из них предназначена для определенной ситуации и реализует тот или иной нетривиальный аспект подсистемы модулей. Тем не менее рекомендуется получить хотя бы общее представление об их назначении, поскольку в ходе дальнейшего знакомства с Java вы столкнетесь с тем, что в некоторых случаях они обеспечивают очень удобные и элегантные решения.

Открытые модули

Как уже упоминалось ранее, по умолчанию классы, содержащиеся в пакетах модуля, доступны только при их явном экспорте с помощью ключевого слова `exports`. Обычно это именно то, что нужно, но бывают ситуации, когда полезно предоставить доступ ко всем пакетам модуля на этапе выполнения, независимо от того, экспортируется пакет или нет. Для этих целей создается ***открытый модуль***, объявляемый с помощью инструкции `open module`.

```
open module имя_модуля {
   // определение модуля
}
```

В открытом модуле типы, определенные во всех пакетах, доступны на этапе выполнения приложения. Но учтите, что только явно экспортируемые пакеты доступны на этапе компиляции. Таким образом, модификатор `open` влияет только на доступность во время выполнения программы.

Открытый модуль призван сделать доступными пакеты модуля через рефлексию. *Рефлексия* — это механизм, позволяющий программе анализировать код на этапе выполнения. Тема рефлексии выходит за рамки данной книги, хотя сам по себе этот механизм может играть важную роль для некоторых типов программ, получающих доступ к сторонним библиотекам на этапе выполнения.

Примечание

Подробнее о рефлексии можно прочитать в книге *Java. Полное руководство, 10-е издание*.

Инструкция opens

Модуль может открыть доступ к определенному пакету на этапе выполнения для других модулей или предоставить доступ к нему с помощью рефлексии вместо того, чтобы открывать весь модуль. Для этого предназначена инструкция `opens`:

```
opens имя_пакета;
```

где *имя_пакета* определяет открываемый пакет. Можно также добавить предложение `to`, задающее модули, для которых открывается пакет.

Учтите, что инструкция `opens` не предоставляет доступ на этапе компиляции. Она используется только для открытия пакета на этапе выполнения и доступа с помощью рефлексии. И еще: инструкция `opens` не может использоваться в открытом модуле. Не забывайте, что все пакеты в открытом модуле уже открыты.

Инструкция requires static

Как известно, инструкция `requires` определяет зависимость, которая по умолчанию проверяется на этапе компиляции и на этапе выполнения. Данное требование можно ослабить таким образом, чтобы модуль не требовался на этапе выполнения. Это реализуется благодаря добавлению модификатора `static` в инструкцию `requires`. Например, следующая инструкция означает, что модуль `mymod` требуется только для компиляции, но не на этапе выполнения:

```
requires static mymod;
```

Добавление ключевого слова `static` делает модуль `mymod` необязательным на этапе выполнения. Это может оказаться полезным в ситуациях, когда программа задействует функциональность при ее наличии, но не требует ее.

Дополнительные сведения о модулях

Выше рассматривались основные элементы подсистемы модулей Java. Это средства, которые непосредственно поддерживаются ключевыми словами в языке Java. Соответственно, о них должен иметь элементарное представление каждый Java-программист. Модульная подсистема предлагает также дополнительные средства, с которыми имеет смысл ознакомиться в ходе дальнейшего изучения Java. Лучшим вариантом для такого знакомства являются утилиты `javac` и `java` с расширенными опциями, имеющими отношение к модулям.

Укажем еще ряд направлений для исследования. В версии JDK 9 появилась утилита `jlink`, которая осуществляет сборку модульного приложения в исполняемый образ, включающий только необходимые модули. Это позволяет экономить дисковое пространство и уменьшать время загрузки приложения. Модульное приложение может быть упаковано в файл JAR (*Java Archive*). Это формат

файла, используемый для развертывания приложений. В результате у утилиты `jar` появились опции, поддерживающие работу с модулями. Например, она может распознавать путь к модулю. JAR-файл, содержащий файл `module-info.class`, называется *модульным JAR-файлом*. Читатели, которые интересуются профессиональными приемами работы с модулями, могут поискать информацию о слоях модулей, автоматических модулях и методиках добавления модулей во время компиляции или на этапе выполнения.

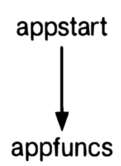
В заключение добавим, что модули будут играть все более важную роль в программировании на Java. Несмотря на необязательность их применения, они предлагают важные преимущества при разработке коммерческих приложений, которые просто нельзя игнорировать. Вполне вероятно, что в недалеком будущем каждый Java-программист будет разрабатывать приложения на основе модулей.

Спросим у эксперта

ВОПРОС. В процессе знакомства с модулями я столкнулся с термином *схема модулей*. Что он означает?

ОТВЕТ. Во время компиляции компилятор разрешает отношения зависимости между модулями путем создания *схемы модулей*, которая отражает эти зависимости. Процесс гарантирует разрешение *всех зависимостей*, включая те, которые имеют косвенный характер. Например, если модуль A требует модуль B, а модуль B требует модуль C, то схема модулей будет включать модуль C, даже если модуль A не использует его напрямую.

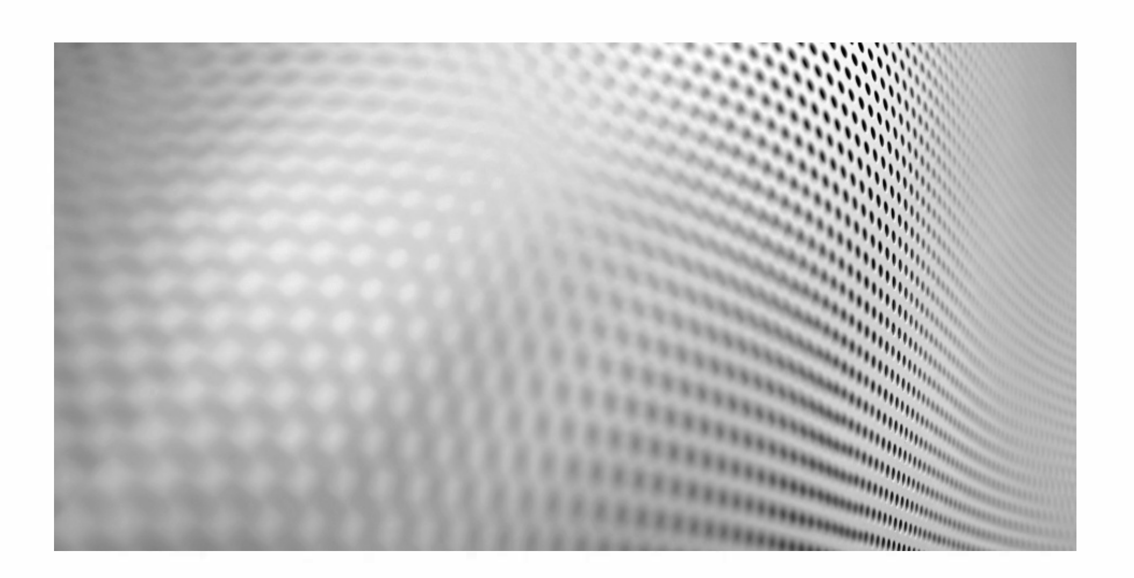
Схему модулей можно представить в виде диаграммы, на которой стрелками показаны отношения между модулями. Вы наверняка столкнетесь с такими диаграммами в ходе дальнейшего изучения темы модулей в Java. Ниже показан простой пример. Это диаграмма для первого приложения данной главы. (Поскольку модуль `java.base` подключается автоматически, он не представлен на диаграмме.)



В Java стрелочки указывают направление от зависимого модуля к требуемому модулю, т.е. на диаграмме можно увидеть, какие модули получают доступ к другим модулям. По правде говоря, из-за сложности большинства коммерческих приложений визуальные схемы модулей целесообразно строить лишь для очень маленьких приложений.

Вопросы и упражнения для самопроверки

1. В широком смысле модули помогают определить зависимость одной единицы кода от другой. Верно ли это утверждение?
2. Какое ключевое слово применяется для объявления модуля?
3. Ключевые слова, поддерживающие модули, являются контекстно-зависимыми. Объясните, что это значит.
4. Что представляет собой файл `module-info.java` и в чем его важность?
5. Какое ключевое слово используется для объявления зависимости одного модуля от другого?
6. Чтобы открытые компоненты пакета стали доступны за пределами модуля, в котором они содержатся, пакет следует указать в инструкции ________.
7. Почему путь к модулю важно указывать при компиляции или выполнении модульного приложения?
8. Что делает инструкция `requires transitive`?
9. Инструкция `exports` экспортирует модуль или пакет?
10. Какая ошибка может возникнуть, если в первом примере модуля удалить строку `exports appfuncs.simplefuncs;` из файла `module-info` для модуля `appfuncs`, а затем попытаться скомпилировать программу?
11. Какие ключевые слова применяются для работы с модульными службами?
12. Служба определяет общую функциональность программной единицы с помощью интерфейса или абстрактного класса. Верно ли это утверждение?
13. Провайдер службы ____________ службу.
14. Какой класс используется для загрузки службы?
15. Может ли модульная зависимость быть необязательной на этапе выполнения? Если да, то каким образом?
16. Вкратце объясните, для чего используются инструкции `open` и `opens`.



Глава 16

Введение в Swing

В этой главе...

- Происхождение и философия Swing
- Компоненты и контейнеры Swing
- Основные сведения о менеджерах компоновки
- Создание, компиляция и выполнение простого Swing-приложения
- Основы обработки событий
- Использование компонента `JButton`
- Работа с компонентом `JTextField`
- Создание флажков `JCheckBox`
- Работа с компонентом `JList`
- Использование анонимных внутренних классов и лямбда-выражений для обработки событий

Все программы, которые приводились в качестве примеров в предыдущих главах, были консольными. Это означает, что в них не использовался графический интерфейс пользователя (Graphical User Interface — GUI). Консольные программы весьма удобны для изучения основ Java и эффективно используются в целом ряде специализированых приложений, например в серверном коде, но в большинстве реальных приложений имеется графический пользовательский интерфейс. Во время написания данной книги наиболее популярным средством для создания подобных Java-приложений была библиотека Swing.

Библиотека Swing предоставляет коллекцию классов и интерфейсов, поддерживающих богатый набор визуальных компонентов, таких как кнопки, поля ввода текста, полосы прокрутки, флажки, деревья узлов и таблицы. Наличие столь широкой палитры элементов управления позволяет создавать чрезвычайно эффективные и вместе с тем простые в использовании графические интерфейсы. Учитывая необычайную популярность библиотеки Swing, ее можно с уверенностью отнести к категории средств, с которыми должен быть знаком любой разработчик, пишущий программы на Java.

Необходимо с самого начала подчеркнуть, что тема Swing очень обширна, и для ее подробного обсуждения понадобилась бы отдельная книга. Поэтому в данной главе мы коснемся лишь самых важных вопросов. Однако и этого будет достаточно для того, чтобы вы получили общее представление о том, что такое библиотека Swing, ознакомились с историей ее создания, основными концепциями и философией проектирования. В этой главе рассматриваются

пять наиболее часто используемых компонентов (элементов GUI), создаваемых средствами Swing: ярлыки, кнопки, текстовые поля, флажки и списки. Завершает главу демонстрационный пример, в котором показано, как создавать аплеты на основе Swing. Несмотря на то что ниже описана лишь небольшая часть инструментальных средств Swing, изучив их, вы сможете самостоятельно создавать несложные программы с GUI-поддержкой. Кроме того, это подготовит вас к последующему более детальному изучению всех возможностей Swing.

Прежде чем продолжить, важно упомянуть о том, что в JDK 8 появилась библиотека JavaFX, которая создавалась специально для поддержки графического пользовательского интерфейса в программах на Java. В ней реализован весьма эффективный, тщательно продуманный и гибкий подход, позволивший значительно упростить создание визуально привлекательных графических интерфейсов. Поэтому библиотека JavaFX может по праву считаться платформой будущего. Учитывая данное обстоятельство, в книгу была дополнительно включена глава 17, содержащая краткий обзор этой библиотеки. Можно ожидать, что в будущем программы на Java будут разрабатываться с использованием одновременно обеих библиотек — Swing и JavaFX.

Происхождение и философия Swing

В ранних версиях Java средства Swing отсутствовали. Их появление было обусловлено стремлением устранить недостатки, свойственные оригинальной подсистеме GUI в Java, реализованной в виде библиотеки AWT (Abstract Window Toolkit). Библиотека AWT содержит базовый набор компонентов, поддерживающих создание вполне работоспособных, но ограниченных по своим возможностям графических пользовательских интерфейсов. Ограниченность библиотеки AWT объясняется, в частности, тем, что различные ее визуальные компоненты транслируются в соответствующие платформенно-зависимые эквиваленты, так называемые *равноправные компоненты* (peers). Отсюда следует, что внешний вид компонентов AWT определяется не средствами Java, а платформой. Поскольку в компонентах AWT используются ресурсы в виде машинно-зависимого кода, их называют *тяжеловесными* (heavyweight).

Использование машинно-зависимых равноправных компонентов порождает ряд проблем. Во-первых, из-за отличий в операционных системах компоненты могут выглядеть и даже вести себя по-разному на различных платформах, что нарушает основополагающий принцип Java "написано однажды, работает везде". Во-вторых, внешний вид каждого компонента остается фиксированным, и изменить его очень трудно (причина та же — зависимость от конкретной платформы). И в-третьих, применение тяжеловесных компонентов влечет за собой ряд новых ограничений. В частности, тяжеловесный компонент всегда имеет прямоугольную форму и является непрозрачным.

Вскоре после выпуска первоначальной версии Java стало очевидным, что ограничения AWT настолько серьезны, что для их преодоления требуется

совершенно иной подход. В итоге в 1997 году появилась библиотека компонентов Swing, включенная в состав набора библиотек классов JFC (Java Foundation Classes). Первоначально библиотека Swing использовалась в версии Java 1.1 как отдельная библиотека. Но в версии Java 1.2 средства Swing (как, впрочем, и остальные элементы JFC) были полностью интегрированы в Java.

Swing устраняет ограничения, присущие компонентам AWT, благодаря использованию двух основных средств: *легковесных компонентов* и *подключаемых стилей оформления*. Несмотря на то что программисту почти не приходится использовать эти средства напрямую, именно они составляют фундамент философии проектирования, заложенной в Swing, и в значительной мере обуславливают возможности и удобство использования этой библиотеки. Рассмотрим каждое из них в отдельности.

За небольшим исключением все компоненты Swing являются *легковесными*. Это означает, что они написаны полностью на Java и не зависят от конкретной платформы, поскольку не опираются на платформенно-зависимые равноправные компоненты. Легковесные компоненты обладают рядом существенных преимуществ, к числу которых относятся эффективность и гибкость. Например, легковесный компонент может быть прозрачным, а его форма может отличаться от прямоугольной. Легковесные компоненты не транслируются в платформенно-зависимые равноправные компоненты, и поэтому их внешний вид определяет библиотека Swing, а не базовая операционная система. Следовательно, элементы пользовательского интерфейса, созданные средствами Swing, выглядят одинаково на разных платформах.

Благодаря тому что каждый компонент Swing визуализируется кодом Java, а не платформенно-зависимыми равноправными компонентами, становится возможным раздельное управление внешним видом компонента и логикой его функционирования, и именно эту задачу решает Swing. Такое разделение обеспечивает значительное преимущество: оно позволяет изменить внешний вид компонента, не затрагивая другие его свойства. Иными словами, появляется возможность подключать новый стиль оформления к компоненту, не создавая никаких побочных эффектов в коде, использующем данный компонент.

Java предоставляет различные стили оформления, такие как "металлик" и Nimbus, доступные каждому пользователю Swing. Металлический стиль также называют *стилем оформления Java*. Это платформенно-независимый стиль оформления, доступный во всех средах выполнения программ на Java. Он же применяется по умолчанию, поэтому именно он и будет использоваться в примерах данной главы.

Реализация подключаемых стилей оформления в Swing стала возможной благодаря тому, что при создании Swing использовался видоизмененный вариант классической архитектуры *модель–представление–контроллер (MVC)*. В терминологии MVC *модель* соответствует информации о состоянии, ассоциированном с компонентом. Например, в случае флажка модель содержит поле,

указывающее на состояние флажка. *Представление* определяет, как выглядит компонент на экране, включая любые аспекты представления, на которые может влиять текущее состояние модели. *Контроллер* определяет реакцию компонента на действия пользователя. Так, если пользователь щелкнет мышью на флажке, контроллер отреагирует, изменив модель таким образом, чтобы отразить выбор пользователя (установку или сброс флажка). В ответ на действия пользователя обновляется и представление. Разделение компонента на модель, представление и контроллер позволяет добиться того, что особенности реализации одной из этих составляющих не будут влиять на две другие. Например, в некоторых реализациях представления один и тот же компонент может отображаться разными способами, а модель и контроллер — оставаться без изменения.

Несмотря на всю концептуальную привлекательность архитектуры MVC и лежащих в ее основе принципов, для компонентов Swing разделение функций между представлением и контроллером не обеспечило заметных преимуществ. В связи с этим в Swing используется видоизмененный вариант MVC, в котором представление и контроллер объединены в единую логическую сущность, называемую *делегатом пользовательского интерфейса*. Поэтому принятый в Swing подход называется архитектурой *модель — делегат*, или архитектурой с *отделяемой моделью*. Таким образом, компоненты Swing нельзя рассматривать как классическую реализацию архитектуры MVC, хотя их архитектура и опирается на нее. В процессе разработки вам не придется иметь дело непосредственно с моделями или делегатами пользовательского интерфейса, но они будут незримо присутствовать в создаваемых вами программах.

Прорабатывая материал главы, вы обнаружите, что библиотека Swing необычайно проста в применении, хотя и основана на довольно сложных принципах проектирования. Одним из аргументов в пользу Swing служит то обстоятельство, что эта библиотека улучшает управляемость такого сложного процесса, как построение пользовательского интерфейса. Это дает разработчикам возможность сосредоточить основное внимание на самом графическом интерфейсе приложения, не отвлекаясь на детали его реализации.

Спросим у эксперта

ВОПРОС. Как отмечалось выше, библиотека Swing определяет внешний вид современных Java-приложений с графическим пользовательским интерфейсом. Означает ли это, что средства Swing фактически заменили AWT?

ОТВЕТ. Нет, не означает. Swing нельзя рассматривать как полную замену AWT. Напротив, библиотека Swing была построена на основе библиотеки AWT, а следовательно, AWT остается важной составной частью Java. Для изучения материала главы знать AWT не обязательно, но если вы хотите в полной мере овладеть Swing, то вам придется основательно разобраться в структуре и возможностях AWT.

Компоненты и контейнеры

Графический пользовательский интерфейс Swing состоит из двух ключевых элементов: *компонентов* и *контейнеров*. Подобное разделение во многом условно, поскольку все контейнеры одновременно являются компонентами. Различие между ними кроется в их предполагаемом назначении. В общепринятом понимании компонент — это независимый визуальный элемент управления (например, кнопка или поле ввода текста), тогда как контейнер может включать в себя несколько компонентов. Следовательно, контейнер — это особая разновидность компонента, предназначенная для хранения других компонентов. Более того, чтобы отобразить компонент на экране, его следует поместить в контейнер. Поэтому в графическом интерфейсе Swing должен иметься хотя бы один контейнер. А так как контейнеры одновременно являются компонентами, то один контейнер может содержать в себе другой. Это дает возможность сформировать так называемую *иерархию включения*, на вершине которой находится *контейнер верхнего уровня*.

Компоненты

Подавляющее большинство компонентов Swing являются производными от класса `JComponent`. (Единственное исключение из этого правила — четыре контейнера верхнего уровня, которые будут описаны в следующем разделе.) В классе `JComponent` реализуются функциональные возможности, общие для всех компонентов. Например, в нем поддерживаются подключаемые стили оформления. Этот класс наследует свойства классов `Container` и `Component` из библиотеки AWT. Таким образом, компоненты Swing создаются на основе компонентов AWT и совместимы с ними.

Все компоненты Swing представляются классами, находящимися в пакете `javax.swing`. В приведенной ниже таблице перечислены имена классов всех компонентов Swing (включая компоненты, используемые как контейнеры).

JApplet (не рекомендуется в JDK 9)	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JTextPane
JTogglebutton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Как видите, имена всех классов начинаются с буквы "J". Например, метке соответствует класс JLabel, кнопке — класс JButton, флажку — класс JCheckBox.

Как уже отмечалось, в рамках данной книги нет возможности рассмотреть все компоненты Swing, — для этого потребовалась бы отдельная книга. Но в этой главе будут представлены пять наиболее часто используемых компонентов: JLabel, JButton, JTextField, JCheckBox и JList. Разобравшись в том, как они работают, вам будет легче освоить другие компоненты.

Контейнеры

В Swing определены два типа контейнеров. К первому типу относятся следующие контейнеры верхнего уровня: JFrame, JApplet, JWindow и JDialog. (Контейнер JApplet, который поддерживает аплеты Swing, не рекомендуется к применению в JDK 9.) Эти контейнеры не наследуют класс JComponent, но они наследуют классы Component и Container библиотеки AWT. В отличие от других, легковесных компонентов Swing, контейнеры верхнего уровня являются тяжеловесными. Именно поэтому они образуют отдельную группу в библиотеке Swing.

Как следует из названия, контейнеры верхнего уровня должны находиться на вершине иерархии контейнеров и не могут содержаться в других контейнерах. Более того, любая иерархия должна начинаться именно с контейнера верхнего уровня. В прикладных программах чаще всего используется контейнер типа JFrame.

Контейнеры второго типа являются легковесными и наследуются от класса JComponent. В качестве примера легковесных контейнеров можно привести классы JPanel, JScrollPane и JRootPane. Легковесные контейнеры могут содержаться в других контейнерах и поэтому нередко используются для объединения группы взаимосвязанных компонентов.

Панели контейнеров верхнего уровня

В каждом контейнере верхнего уровня определен набор *панелей*. На вершине иерархии находится корневая панель — экземпляр класса JRootPane, который представляет собой легковесный контейнер, предназначенный для управления другими панелями. Он также позволяет управлять строкой меню. Корневая панель включает в себя *"стеклянную" панель*, *панель содержимого* и *многослойную панель*.

"Стеклянная" (иначе прозрачная) панель является панелью верхнего уровня. Она располагается поверх всех остальных панелей и полностью покрывает их. Прозрачная панель позволяет управлять событиями мыши, относящимися ко всему контейнеру (а не к отдельным элементам управления), и выполнять операции рисования поверх любого другого компонента. В большинстве случаев у вас не будет возникать необходимость в непосредственном использовании

прозрачной панели. Многослойная панель позволяет задавать глубину расположения компонентов, определяя порядок перекрытия одних компонентов другими. (Таким образом, многослойная панель позволяет упорядочивать компоненты по координате Z, хотя это требуется не так уж часто.) В состав многослойной панели входит панель содержимого и (необязательно) строка меню. Несмотря на то что прозрачная и многослойная панели являются неотъемлемыми частями контейнера верхнего уровня и выполняют важные функции, их действия по большей части скрыты не только от пользователей, но и от разработчиков прикладных программ.

Ваше приложение в основном будет взаимодействовать с панелью содержимого, в которую добавляются визуальные компоненты. Иными словами, добавляя компонент, например кнопку, в контейнер верхнего уровня, вы на самом деле добавляете его на панель содержимого.

Менеджеры компоновки

Прежде чем приступить к написанию программ средствами Swing, вам необходимо получить хотя бы общее представление о *менеджерах компоновки*. Менеджер компоновки управляет размещением компонентов в контейнере. В Java определено несколько таких менеджеров. Большинство из них входит в состав AWT (т.е. в пакет `java.awt`), но Swing предоставляет также ряд дополнительных менеджеров компоновки. Все менеджеры компоновки являются экземплярами классов, реализующих интерфейс `LayoutManager`. (Некоторые из менеджеров компоновки реализуют интерфейс `LayoutManager2`.) Ниже перечислен ряд менеджеров компоновки, которые доступны для разработчиков, использующих библиотеку Swing.

`FlowLayout`	Простой менеджер компоновки, размещающий компоненты слева направо и сверху вниз. (Для некоторых региональных настроек компоненты располагаются справа налево.)
`BorderLayout`	Располагает компоненты по центру или по краям контейнера. Этот менеджер принят по умолчанию для панели содержимого
`GridLayout`	Располагает компоненты в ячейках сетки, как в таблице
`GridBagLayout`	Располагает компоненты разных размеров в ячейках сетки с регулируемыми размерами
`BoxLayout`	Располагает компоненты в вертикальном или горизонтальном направлении
`SpringLayout`	Располагает компоненты с учетом ряда ограничений

Менеджеры компоновки, как и многие другие компоненты Swing, невозможно рассмотреть во всех подробностях в одной главе. Поэтому остановимся только на двух из них: `BorderLayout` и `FlowLayout`.

Для панели содержимого менеджером компоновки по умолчанию является `BorderLayout`. Этот менеджер определяет в составе контейнера пять областей, в которые могут помещаться компоненты. Первая область располагается посредине и называется центральной. Остальные четыре располагаются в соответствии со сторонами света и носят такие названия: северная, южная, восточная и западная. По умолчанию компонент, добавляемый на панель содержимого, располагается в центральной области. Для того чтобы расположить компонент в другой области, следует указать ее имя.

Несмотря на то что возможностей, предоставляемых менеджером компоновки `BorderLayout`, зачастую оказывается достаточно, иногда возникает потребность в других менеджерах компоновки. К числу самых простых относится менеджер компоновки `FlowLayout`, который размещает компоненты построчно: слева направо и сверху вниз. Заполнив текущую строку, он переходит к следующей. Такая компоновка предоставляет лишь ограниченный контроль над расположением компонентов, хотя и проста в применении. Однако при изменении размеров контейнера расположение компонентов может измениться.

Первая простая Swing-программа

Программы, создаваемые средствами Swing, отличаются от консольных программ, примеры которых были рассмотрены ранее. Swing-программы не только используют набор компонентов Swing для обработки взаимодействия с пользователем, но и удовлетворяют особым требованиям, связанным с управлением потоками. Для того чтобы стала понятнее структура Swing-программы, лучше всего обратиться к конкретному примеру.

Примечание

Программы Swing, рассматриваемые в этой главе, называются настольными приложениями. Ранее на основе Swing создавались аплеты, но поскольку, начиная с JDK 9, аплеты не рекомендуются к применению, они не рассматриваются в книге.

Несмотря на то что рассматриваемый здесь пример программы довольно прост, он наглядно демонстрирует один из приемов написания Swing-приложений. В данной программе используются два компонента Swing: классы `JFrame` и `JLabel`. Класс `JFrame` представляет собой контейнер верхнего уровня, нередко используемый в Swing-приложениях, а класс `JLabel` — это компонент Swing, с помощью которого создается метка (надпись), используемая для вывода информации. Метка является самым простым компонентом Swing, поскольку она не реагирует на действия пользователя, а только отображает информацию. Контейнер `JFrame` служит для размещения экземпляра компонента `JLabel`. С помощью метки отображается короткое текстовое сообщение.

```
// Простая Swing-программа

import javax.swing.*; ◄———— Swing-программы должны импортировать
                            пакет javax.swing
class SwingDemo {

   SwingDemo() {                                   Создание контейнера

      // Создать новый контейнер JFrame
      JFrame jfrm = new JFrame("Простое приложение Swing"); ◄——

      // Установить начальные размеры фрейма
      jfrm.setSize(275, 100); ◄———— Настройка размеров контейнера

      // Завершить работу программы, когда пользователь
      // закрывает приложение
      jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); ◄—— Прекращение
                                                              выполнения
      // Создать текстовую метку                                при закрытии
      JLabel jlab = new JLabel("Программирование интерфейса
                                с помощью Swing."); ◄—— Добавление метки Swing

      // Добавить метку на панель содержимого
      jfrm.add(jlab); ◄———— Добавление метки на панель содержимого

      // Отобразить фрейм
      jfrm.setVisible(true); ◄———— Отображение фрейма
   }

   public static void main(String args[]) {
      // Создать фрейм в потоке диспетчеризации событий
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            new SwingDemo(); ◄———— Фрейм SwingDemo должен создаваться
         }                          в потоке диспетчеризации событий
      });
   }
}
```

Эта программа компилируется и запускается точно так же, как и любое другое Java-приложение. Для ее компиляции введите в командной строке следующую команду:

```
javac SwingDemo.java
```

Для запуска программы используйте команду.

```
java SwingDemo
```

При выполнении данной программы отображается окно, приведенное на рис. 16.1.

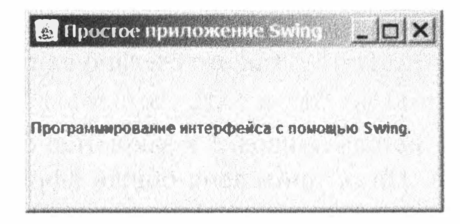


Рис. 16.1. Окно, отображаемое при выполнении программы `SwingDemo`

Построчный анализ первой Swing-программы

Поскольку программа `SwingDemo` иллюстрирует сразу несколько ключевых концепций Swing, проанализируем ее тщательно, строка за строкой. Программа начинается с команды импорта пакета.

```
import javax.swing.*;
```

Этот пакет содержит компоненты и модели, используемые библиотекой Swing. В частности, он определяет классы, реализующие метки, кнопки, текстовые поля и меню. Любая программа, задействующая библиотеку Swing, должна включать этот пакет. Начиная с JDK 9 пакет `javax.swing` находится в модуле `java.desktop`.

Далее объявляется класс `SwingDemo` и его конструктор. Именно в конструкторе выполняется большинство действий программы. Он начинается с создания экземпляра класса `JFrame`.

```
JFrame jfrm = new JFrame("Простое приложение Swing");
```

В результате создается контейнер `jfrm`, определяющий прямоугольное окно со строкой заголовка, кнопками закрытия, свертывания, развертывания и восстановления окна, а также с системным меню. Таким образом, данная строка кода создает стандартное окно верхнего уровня. Строка заголовка передается конструктору в качестве параметра.

Размеры окна задаются с помощью следующей строки кода:

```
jfrm.setSize(275, 100);
```

Используемый для этого метод `setSize()` устанавливает размеры окна в пикселях. Ниже приведена общая форма объявления этого метода.

```
void setSize(int ширина, int высота)
```

В нашем примере ширина окна составляет 275 пикселей, высота — 100 пикселей.

По умолчанию при закрытии окна верхнего уровня (например, когда пользователь щелкает на кнопке закрытия) оно удаляется с экрана, но приложение продолжает работать. Иногда такое поведение окна действительно бывает необходимым, но для большинства случаев оно не подходит. Чаще всего требуется, чтобы закрытие окна сопровождалось завершением работы приложения. Этого

можно добиться несколькими способами. Самый простой из них — вызов метода `setDefaultCloseOperation()`, как это сделано в программе.

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Выполнение данного метода приводит к закрытию окна и завершению работы всего приложения. Ниже приведена общая форма объявления метода `setDefaultCloseOperation()`.

```
void setDefaultCloseOperation(int действие)
```

Значение параметра *действие* определяет, что именно должно произойти при закрытии окна. Кроме константы `JFrame.EXIT_ON_CLOSE`, данному методу можно передавать следующие константы.

```
JFrame.DISPOSE_ON_CLOSE
JFrame.HIDE_ON_CLOSE
JFrame.DO_NOTHING_ON_CLOSE
```

Имена констант отражают выполняемые действия. Все эти константы определены в интерфейсе `WindowConstants` (пакет `javax.swing`), реализуемом классом `JFrame`.

В следующей строке кода создается компонент `JLabel`:

```
JLabel jlab = new JLabel("Программирование интерфейса
                         с помощью Swing.");
```

Компонент `JLabel` — самый простой в использовании среди всех компонентов Swing, поскольку он не предполагает обработку событий, связанных с действиями пользователя, а только отображает информацию: текст, изображение или и то и другое. Метка, созданная в данной программе, содержит только текст, который передается ее конструктору.

Следующая строка кода добавляет метку на панель содержимого фрейма:

```
jfrm.add(jlab);
```

Как уже отмечалось, все контейнеры верхнего уровня имеют панель содержимого, в которой размещаются компоненты. Следовательно, чтобы добавить компонент во фрейм, его следует добавить на панель содержимого. Это достигается путем вызова метода `add()` для ссылки на экземпляр класса `JFrame` (переменная `jfrm`). Существует несколько вариантов метода `add()`, но чаще других используется такой вариант:

```
Component add(Component компонент)
```

По умолчанию панель содержимого, ассоциированная с контейнером `JFrame`, использует граничную компоновку. В приведенном выше варианте метода `add()` компонент (в данном случае метка) добавляется по центру. Другие варианты метода `add()` позволяют задавать для компоновки одну из граничных областей. Когда компонент добавляется в центральную область, его размер автоматически изменяется для того, чтобы он мог уместиться по центру.

Последняя инструкция в конструкторе класса SwingDemo делает окно видимым:

```
jfrm.setVisible(true);
```

Общий синтаксис объявления метода setVisible() выглядит так:

```
void setVisible(boolean флаг)
```

Если параметр *флаг* принимает значение true, то окно отображается на экране, в противном случае оно остается скрытым. По умолчанию фрейм (объект типа JFrame) не виден, поэтому для его отображения требуется вызов метода setVisible(true).

В методе main() создается объект типа SwingDemo, отображающий окно и метку на экране. Особого внимания заслуживает способ вызова конструктора класса SwingDemo.

```
SwingUtilities.invokeLater(new Runnable() {
   public void run() {
      new SwingDemo();
   }
});
```

Объект типа SwingDemo создается здесь не в основном потоке приложения, а в *потоке диспетчеризации событий*, и вот почему. Обычно программы Swing управляются событиями. Например, когда пользователь активизирует GUI-компонент, генерируется соответствующее событие. Событие передается приложению путем вызова обработчика событий, определяемого приложением. Однако данный обработчик выполняется не в главном потоке приложения, а в потоке диспетчеризации событий, предоставляемом библиотекой Swing. Таким образом, несмотря на то что обработчики событий определяются в программе, они выполняются в потоке, создаваемом вне программы. Чтобы избежать возможных проблем (например, попытки двух потоков одновременно обновить один и тот же компонент), все компоненты GUI библиотеки Swing должны создаваться и обновляться в потоке диспетчеризации событий, а не в главном потоке приложения. Однако метод main() выполняется в основном потоке и поэтому не может напрямую создавать экземпляры класса SwingDemo. Что он может, так это создать объект типа Runnable, выполняющийся в потоке диспетчеризации событий, и поручить создание GUI этому объекту.

Для создания кода GUI в потоке диспетчеризации событий необходимо использовать один из двух методов, определенных в классе SwingUtilities: invokeLater() или invokeAndWait(). Эти методы объявляются следующим образом.

```
static void invokeLater(Runnable объект)

static void invokeAndWait(Runnable объект)
     throws InterruptedException, InvocationTargetException
```

Здесь *объект* — это объект типа `Runnable`, метод `run()` которого вызывается в потоке диспетчеризации событий. Различие между этими методами состоит в том, что метод `invokeLater()` сразу же возвращает управление вызывающему методу, тогда как метод `invokeAndWait()` ожидает возврата из метода `run()`. Эти методы можно использовать для вызова метода, создающего графический пользовательский интерфейс приложения Swing, а также во всех других случаях, когда требуется изменить состояние GUI из кода, выполняющегося не в потоке диспетчеризации событий. Как правило, вы будете использовать метод `invokeLater()`, как это сделано в предыдущей программе.

Следует отметить еще одну особенность анализируемой программы: в ней не предусмотрена реакция на события, поскольку компонент `JLabel` пассивный, т.е. не генерирует событий. Однако остальные компоненты генерируют события, на которые программа должна реагировать соответствующим образом, что и будет продемонстрировано далее на конкретных примерах.

Спросим у эксперта

ВОПРОС. Ранее было сказано, что компоненты можно добавлять и в другие области, а не только по центру, и для этого предусмотрены специальные варианты метода `add()`. Нельзя ли рассказать об этом подробнее?

ОТВЕТ. Как уже отмечалось, для размещения компонентов в контейнере менеджер компоновки `BorderLayout` определяет пять областей. Одна из них — центральная (`CENTER`), а остальные четыре располагаются в соответствии со сторонами света и называются северной (`NORTH`), южной (`SOUTH`), восточной (`EAST`) и западной (`WEST`) сторонами. По умолчанию компонент, добавляемый на панель содержимого, располагается по центру. Для того чтобы указать другое расположение компонента, используйте следующий вариант метода `add()`:

```
void add(Component компонент, Object расположение)
```

где параметр *компонент* задает компонент, добавляемый на панель содержимого, а параметр *расположение* определяет область, в которой этот компонент будет находиться. Допускаются следующие значения параметра *расположение*.

```
BorderLayout.CENTER  BorderLayout.EAST  BorderLayout.NORTH
BorderLayout.SOUTH   BorderLayout.WEST
```

Вообще говоря, менеджер компоновки `BorderLayout` чаще всего используется для создания контейнеров `JFrame`, содержащих центральный компонент (или группу компонентов, размещаемых в одном из легковесных контейнеров Swing), а также ассоциированные с ним компоненты верхнего и нижнего колонтитулов. В остальных же случаях более подходящими будут другие менеджеры компоновки Java.

Обработка событий Swing

Как уже упоминалось, в целом программы Swing управляются событиями, т.е. компоненты взаимодействуют с программой с помощью событий. Например, событие, сгенерированное после щелчка пользователя на кнопке, перемещает указатель мыши или выбирает элемент из списка. События также могут генерироваться другими способами. Например, событие будет сгенерировано после завершения отсчета таймера. Как только событие отсылается программе, она реагирует на событие путем вызова обработчика событий. Как видите, обработка событий является важной частью практически всех приложений Swing.

Механизм обработки событий, используемый в Swing, называется *моделью делегирования событий*. Концепция этой модели довольно проста. *Источник* события генерирует событие и отсылает его одному или нескольким *слушателям*. При использовании этого подхода слушатель просто ждет до тех пор, пока не получит событие. Как только событие получено, слушатель обрабатывает и возвращает его. Преимущество этого подхода заключается в том, что логика приложения, ответственная за обработку событий, четко отделена от логики интерфейса пользователя, которая генерирует события. Таким образом интерфейс пользователя может “делегировать” обработку события отдельной части кода. В модели делегирования событий слушатель должен регистрировать источник, чтобы иметь возможность получать события.

Ознакомимся поближе с событиями, источниками и слушателями.

События

В Java *событие* — это объект, который описывает, как изменилось состояние источника события. Оно может быть сгенерировано вследствие взаимодействия пользователя с элементом графического интерфейса или же самой программой. Суперклассом для всех событий является `java.util.EventObject`. Многие события объявлены в пакете `java.awt.event`. События, которые связаны исключительно со Swing, находятся в пакете `javax.swing.event`.

Источники событий

Источник события — это объект, генерирующий событие. Как только источник генерирует событие, он тут же отсылает его всем зарегистрированным слушателям. Поэтому для получения события слушатель должен зарегистрироваться в источнике этого события. В Swing регистрация слушателей в источнике события осуществляется путем вызова метода для исходного объекта события. Обычно для событий используется следующее соглашение о наименовании:

```
public void addТипListener (ТипListener сс)
```

где *Тип* — это название события, а *сс* — ссылка на слушателя события. Например, метод, который регистрирует слушателя события клавиатуры, называется `addKeyListener()`. Метод, который регистрирует перемещение указателя

мыши, называется `addMouseMotionListener()`. Как только происходит событие, оно тут же передается всем зарегистрированным слушателям.

Источник также должен поддерживать метод, который позволяет отменять регистрацию указанного типа события. В Swing при этом используется метод, для которого применяется следующее соглашение о наименовании:

```
public void removeТипListener(ТипListener сс)
```

И снова *Тип* — это название события, а *сс* — ссылка на слушателя события. Например, для удаления слушателя клавиатуры можно воспользоваться методом `removeKeyListener()`.

Методы, которые добавляют или удаляют слушателей, поддерживаются источником, генерирующим события. Например, как будет вскоре показано, класс `JButton` является источником событий `ActionEvents`, свидетельствующих о выполнении какого-либо действия, такого как нажатие кнопки. Таким образом, класс `JButton` поддерживает методы, применяемые для добавления или удаления слушателя действия.

Слушатели событий

Слушатель — это объект, который извещается о произошедшем событии. К данному объекту предъявляются два основных требования. Во-первых, он должен быть зарегистрирован в одном или нескольких источниках для получения определенного типа событий. Во-вторых, он должен реализовать метод для получения и обработки событий.

Методы, которые получают и обрабатывают события Swing, определены в наборах интерфейсов, таких как `java.awt.event` и `javax.swing.event`. Например, интерфейс `ActionListener` определяет метод, который обрабатывает событие `ActionEvent`. Любой объект может принимать и обрабатывать это событие, если поддерживается реализация интерфейса `ActionListener`.

А сейчас сформулируем очень важный принцип: обработчик событий должен выполнять свою работу быстро и сразу же завершаться. В большинстве случаев он не должен участвовать в выполнении длительных операций, поскольку это замедлит работу всего приложения. Для выполнения ресурсоемких операций понадобится отдельный поток.

Классы событий и интерфейсы слушателей

Классы, которые представляют события, определены в ядре механизма обработки событий Swing. В корне иерархии классов событий находится класс `EventObject`, который помещен в пакет `java.util`. Это суперкласс для всех событий в Java. Класс `AWTEvent`, объявленный в пакете `java.awt`, является подклассом `EventObject`. Это суперкласс (прямым или косвенным образом) для всех основанных на AWT событиях, используемых моделью делегирования событий. И хотя Swing использует события AWT, кроме них добавлены еще

собственные события. Как упоминалось ранее, эти события находятся в пакете `javax.swing.event`. Таким образом, Swing поддерживает большое число событий. Но в этой главе будут рассмотрены только три из них. Эти события приведены в следующей таблице наряду с соответствующими слушателями.

Класс события	Описание	Соответствующий слушатель событий
`ActionEvent`	Генерируется при выполнении действия, которое происходит в элементе управления, например щелчок на кнопке	`ActionListener`
`ItemEvent`	Генерируется при выборе элемента, например после щелчка на флажке	`ItemListener`
`ListSelectionEvent`	Генерируется при изменении выделения в списке	`ListSelectionListener`

Следующие примеры иллюстрируют общий подход, применяемый при обработке указанных событий. Те же самые действия будут выполняться и при обработке событий Swing в целом. Как вы вскоре поймете, процесс обработки событий достаточно прямолинеен и прост.

Использование компонента JButton

Одним из наиболее часто используемых визуальных элементов управления Swing является кнопка. Кнопка Swing — это экземпляр класса `JButton`. Класс `JButton` наследует абстрактный класс `AbstractButton`, который определяет функциональность, общую для всех кнопок. На кнопке может отображаться текст, картинка или и то и другое, но в этой книге рассматриваются только кнопки с надписями.

Класс `JButton` предоставляет несколько конструкторов. Мы будем использовать конструктор следующего вида:

```
JButton(String сообщение)
```

где *сообщение* определяет строку, которая должна отображаться в виде надписи на кнопке.

После щелчка на кнопке генерируется событие `ActionEvent`. Класс `ActionEvent` определен в библиотеке AWT, но используется и в библиотеке Swing. В классе `JButton` предоставляются методы, позволяющие зарегистрировать слушателя событий или отменить его регистрацию.

```
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)
```

Здесь параметр *al* задает объект, который будет получать уведомления о наступлении событий. Этот объект должен представлять собой экземпляр класса, реализующего интерфейс ActionListener.

В интерфейсе ActionListener определен только один метод: actionPerformed(). Ниже приведен его синтаксис:

```
void actionPerformed(ActionEvent ae)
```

Данный метод вызывается после щелчка на кнопке. Иными словами, он является обработчиком события щелчка на кнопке. Реализуя метод actionPerformed(), необходимо позаботиться о том, чтобы он быстро реагировал на событие и сразу возвращал управление. В отношении обработчиков событий необходимо руководствоваться следующим общим правилом: они не должны вовлекаться в длительные операции, поскольку это будет замедлять работу приложения в целом. Если же обработка события предполагает действия, требующие времени, их следует выполнять в отдельном потоке, специально создаваемом для этой цели.

С помощью объекта типа ActionEvent, передаваемого методу actionPerformed(), можно получить важные сведения о событии щелчка на кнопке. В данной главе для этой цели будет использоваться строка *команды действия*, связанная с кнопкой. По умолчанию именно эта строка отображается на кнопке. Чтобы получить команду действия, следует вызвать метод getActionCommand() для объекта события. Этот метод объявляется следующим образом:

```
String getActionCommand()
```

Команда действия идентифицирует кнопку. При наличии в пользовательском интерфейсе приложения нескольких кнопок команда действия позволяет достаточно легко определить, какая из них была выбрана.

Ниже приведен пример программы, в которой показано, как создать кнопку, реагирующую на действия пользователя. Окно, отображаемое данной программой на экране, представлено на рис. 16.2.

```
// Демонстрация создания кнопки и обработки событий действий

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class ButtonDemo implements ActionListener {

   JLabel jlab;

   ButtonDemo() {

      // Создать новый контейнер JFrame
      JFrame jfrm = new JFrame("Пример кнопки");
```

```
        // Задать объект FlowLayout для менеджера компоновки
        jfrm.setLayout(new FlowLayout());

        // Задать исходные размеры фрейма
        jfrm.setSize(220, 90);

        // Прекратить работу программы, если
        // пользователь закрывает приложение
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создать две кнопки
        JButton jbtnUp = new JButton("Отпущена");
        JButton jbtnDown = new JButton("Нажата");

        // Добавить слушатели действий
        jbtnUp.addActionListener(this);
        jbtnDown.addActionListener(this);

        // Добавить кнопки на панель содержимого
        jfrm.add(jbtnUp);
        jfrm.add(jbtnDown);

        // Создать метку
        jlab = new JLabel("Нажать кнопку.");

        // Добавить метку во фрейм
        jfrm.add(jlab);

        // Отобразить фрейм
        jfrm.setVisible(true);
    }

    // Обработка событий кнопки
    public void actionPerformed(ActionEvent ae) {
        if(ae.getActionCommand().equals("Отпущена"))
            jlab.setText("Кнопка отпущена.");
        else
            jlab.setText("Кнопка нажата.");
    }

    public static void main(String args[]) {
        // Создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new ButtonDemo();
            }
        });
    }
}
```

Создать две кнопки

Добавить слушателей действий для кнопок

Добавление кнопок на панель содержимого

Обработка событий кнопки

Использование команды действия для определения состояния кнопки

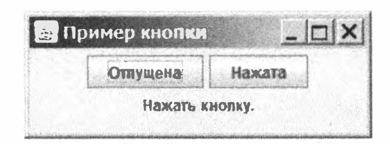


Рис. 16.2. Результат выполнения программы `ButtonDemo`

Проанализируем, что нового появилось в этой программе. Прежде всего, теперь программа импортирует два пакета: `java.awt` и `java.awt.event`. Пакет `java.awt` необходим потому, что он содержит класс менеджера компоновки `FlowLayout`, а пакет `java.awt.event` — потому, что в нем определены интерфейс `ActionListener` и класс `ActionEvent`.

Далее в программе объявляется класс `ButtonDemo`, который реализует интерфейс `ActionListener`. Это означает, что объекты типа `ButtonDemo` могут быть использованы для получения событий действий. Затем объявляется ссылка на объект типа `JLabel`. Она будет использована в методе `actionPerformed()` для отображения сведений о том, какая именно кнопка была нажата.

Конструктор класса `ButtonDemo` начинается с создания контейнера `jfrm` типа `JFrame`. Затем в качестве менеджера компоновки для панели содержимого контейнера `jfrm` устанавливается `FlowLayout`:

```
jfrm.setLayout(new FlowLayout());
```

Как уже отмечалось, по умолчанию на панели содержимого в качестве менеджера компоновки используется `BorderLayout`, но для многих приложений лучше подходит менеджер компоновки `FlowLayout`. Он располагает компоненты построчно: слева направо и сверху вниз. После заполнения текущей строки менеджер компоновки переходит к следующей. Такая компоновка предоставляет лишь ограниченный контроль над расположением компонентов, зато она проста в использовании. Однако при изменении размеров контейнера расположение компонентов может измениться.

После установки размеров фрейма и определения операции, выполняемой при закрытии окна, в конструкторе `ButtonDemo()` создаются две кнопки.

```
JButton jbtnUp = new JButton("Отпущена");
JButton jbtnDown = new JButton("Нажата");
```

На первой кнопке отображается надпись Отпущена, на второй — Нажата.

Далее экземпляр класса `ButtonDemo`, для ссылки на который используется ключевое слово `this`, добавляется в качестве слушателя событий для кнопок с помощью следующих строк кода.

```
jbtnUp.addActionListener(this);
jbtnDown.addActionListener(this);
```

В результате выполнения этого кода объект, создающий кнопки, будет получать также уведомления об их нажатии.

Всякий раз, когда кнопка нажимается, генерируется событие действия, о котором зарегистрированные слушатели уведомляются посредством вызова метода `actionPerformed()`. Объект типа `ActionEvent`, представляющий событие кнопки, передается этому методу в качестве параметра. В программе `ButtonDemo` это событие передается следующей реализации метода `actionPerformed()`.

```
// Обработка событий кнопки
public void actionPerformed(ActionEvent ae) {
   if(ae.getActionCommand().equals("Отпущена"))
      jlab.setText("Кнопка отпущена.");
   else
      jlab.setText("Кнопка нажата.");
}
```

Событие передается с помощью параметра `ae`. В теле метода для получения команды действия, которая соответствует кнопке, сгенерировавшей событие, вызывается метод `getActionCommand()`. (Напомним: по умолчанию команда действия совпадает с текстом, отображаемым на кнопке.) В зависимости от содержимого строки, представляющей команду действия, устанавливается текст метки, указывающий на то, какая именно кнопка была нажата.

Следует также иметь в виду, что, как отмечалось ранее, метод `actionPerformed()` вызывается в потоке диспетчеризации событий. Он должен возвращать управление как можно быстрее, чтобы не замедлять работу приложения.

Работа с компонентом `JTextField`

К числу широко используемых компонентов Swing относится также компонент `JTextField`, который дает пользователю возможность вводить и редактировать текстовые строки. Компонент `JTextField` является подклассом, производным от абстрактного класса `JTextComponent`, который выступает в качестве суперкласса не только для компонента `JTextField`, но и для всех текстовых компонентов вообще. В классе `JTextField` определен ряд конструкторов. Здесь и далее будет использоваться следующий конструктор:

```
JTextField(int столбцы)
```

где *столбцы* — это ширина текстового поля, выраженная в столбцах. Важно понимать, что длина вводимой строки не ограничивается шириной поля, отображаемого на экране, и параметр *столбцы* устанавливает лишь физический размер компонента на экране.

Для завершения ввода текста в поле пользователь нажимает клавишу <Enter>, в результате чего генерируется событие `ActionEvent`. В классе `JTextField` предоставляются методы `addActionListener()` и `removeActionListener()`. Для обработки событий действий необходимо реализовать метод `actionPerformed()`, объявленный в интерфейсе `ActionListener`.

Обработка событий текстового поля осуществляется аналогично обработке событий кнопки, о которых шла речь ранее.

Как и в случае компонента `JButton`, с компонентом `JTextField` связывается конкретная команда действия в виде строки. По умолчанию эта строка соответствует текущему содержимому текстового поля, хотя эта возможность используется редко. Чаще всего вы будете сами задавать фиксированное значение команды действия с помощью метода `setActionCommand()`, который объявляется следующим образом:

```
void setActionCommand(String команда)
```

Строка, передаваемая через параметр *команда*, становится новой командой действия, но при этом текст в поле не меняется. Установленная строка команды действия остается неизменной, независимо от того, какой именно текст вводится в поле. Как правило, к явной установке команды действия прибегают для того, чтобы обеспечить распознавание текстового поля как источника, сгенерировавшего событие действия. Поступать подобным образом приходится в тех случаях, когда фрейм содержит несколько элементов управления, для которых определен общий обработчик событий. Установив команду действия, вы получаете в свое распоряжение удобное средство для различения компонентов. Если этого не сделать, могут возникнуть трудности при распознавании источника события, так как пользователь может ввести в поле произвольный текст, совпадающий с командой действия другого компонента.

Спросим у эксперта

ВОПРОС. Как отмечалось выше, команду действия для поля ввода текста можно явно установить с помощью метода `setActionCommand()`. Можно ли воспользоваться этим методом, чтобы установить команду действия для кнопки?

ОТВЕТ. Да, можно. По умолчанию команда действия для кнопки совпадает с текстом надписи на кнопке. Но можно установить и другое ее значение, воспользовавшись методом `setActionCommand()` класса `JButton`. Он выполняет те же действия, что и одноименный метод класса `JTextField`.

Для того чтобы получить строку, отображаемую в текстовом поле, следует обратиться к экземпляру класса `JTextField` и вызвать метод `getText()`. Объявление этого метода приведено ниже:

```
String getText()
```

Задать текст для компонента `JTextField` можно с помощью метода `setText()`:

```
void setText(String текст)
```

где *текст* — это строка, которая будет помещена в текстовое поле.

Ниже приведен пример программы, демонстрирующий использование компонента JTextField. В окне этой программы содержатся текстовое поле, кнопка и две метки, одна из которых подсказывает пользователю ввести текст в поле. Когда пользователь нажмет клавишу <Enter> (при условии, что фокус ввода находится в поле ввода текста), введенные данные будут извлечены и отображены во второй метке. На кнопке отображается надпись **Обратить**. После щелчка на кнопке содержимое текстового поля преобразуется, и порядок символов в нем меняется на обратный. Результат выполнения данной программы приведен на рис. 16.3.

```
// Использование текстового поля

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class TFDemo implements ActionListener {

   JTextField jtf;
   JButton jbtnRev;
   JLabel jlabPrompt, jlabContents;

   TFDemo() {

      // Создать новый контейнер JFrame
      JFrame jfrm = new JFrame("Использование текстового поля");

      // Задать объект FlowLayout для менеджера компоновки
      jfrm.setLayout(new FlowLayout());

      // Задать исходные размеры фрейма
      jfrm.setSize(240, 120);

      // Прекратить работу программы, если
      // пользователь закрывает приложение

      jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Создать текстовое поле
      jtf = new JTextField(10); ◄——— Создание текстового поля шириной 10 символов

      // Установить команду действия для текстового поля
      jtf.setActionCommand("myTF"); ◄——— Установка команды действия для текстового
                                          поляfield

      // Создать кнопку
      JButton jbtnRev = new JButton("Обратить");

      // Добавить слушателей событий для текстового поля и кнопки
      jtf.addActionListener(this);     ◄——┐ Добавление слушателей событий
      jbtnRev.addActionListener(this); ◄——┘ для текстового поля и кнопки
```

```
      // Создать метки
      jlabPrompt = new JLabel("Введите текст: ");
      jlabContents = new JLabel("");

      // Добавить компоненты на панель содержимого
      jfrm.add(jlabPrompt);
      jfrm.add(jtf);
      jfrm.add(jbtnRev);
      jfrm.add(jlabContents);

      // Отобразить фрейм
      jfrm.setVisible(true);
   }

   // Обработать события действий
    public void actionPerformed(ActionEvent ae) {   ◄── Этот метод обрабатывает
                                                        события кнопки и текстового
                                                        поля
      if(ae.getActionCommand().equals("Обратить")) {   ◄── Используйте команду
         // Была нажата кнопка                             действия для определения
         String orgStr = jtf.getText();                    того, какой компонент
         String resStr = "";                               сгенерировал событие

         // Обратить строку в текстовом поле
         for(int i=orgStr.length()-1; i >=0; i--)
            resStr += orgStr.charAt(i);

         // Сохранить обращенную строку в текстовом поле
         jtf.setText(resStr);
      } else
         // Клавиша <Enter> была нажата в тот момент, когда фокус
         // ввода находился в текстовом поле
         jlabContents.setText("Вы нажали ENTER. Текст: " +
                              jtf.getText());
   }

   public static void main(String args[]) {
      // Создать фрейм в потоке диспетчеризации событий
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            new TFDemo();
         }
      });
   }
}
```

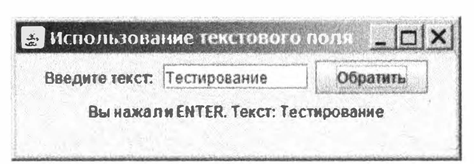


Рис. 16.3. Результат выполнения программы TFDemo

Большая часть исходного кода приведенной выше программы вам уже знакома, но некоторые его фрагменты нуждаются в пояснениях. Прежде всего, обратите внимание на то, что с текстовым полем связывается команда действия "myTF". Эту привязку осуществляет следующая строка кода:

```
jtf.setActionCommand("myTF");
```

После выполнения этого кода строка команды действия всегда будет содержать значение "myTF", независимо от того, какой именно текст введен в поле. Благодаря этому исключаются ситуации, когда команда действия, связанная с текстовым полем, может вступать в конфликт с командой действия, связанной с кнопкой **Обратить**. В методе actionPerformed() команда действия используется для распознавания того компонента, который стал источником события. Если строка команды действия содержит значение "Обратить", то это может означать только одно: событие наступило после щелчка на кнопке **Обратить**. Иначе следует сделать вывод, что событие наступило в результате нажатия пользователем клавиши <Enter> в тот момент, когда фокус ввода находился в текстовом поле.

И наконец, обратите внимание на следующую строку кода в теле метода actionPerformed():

```
jlabContents.setText("Вы нажали ENTER. Текст: " +
                     jtf.getText());
```

Как уже отмечалось, при нажатии клавиши <Enter> в тот момент, когда фокус ввода находится в текстовом поле, генерируется событие ActionEvent, которое пересылается всем зарегистрированным слушателям событий действий с помощью метода actionPerformed(). В программе TFDemo этот метод вызывает метод getText(), извлекая текст, содержащийся в компоненте jtf (текстовое поле). После этого текст отображается с помощью метки, на которую ссылается переменная jlabContents.

Создание флажков с помощью компонента JCheckBox

Если обычные кнопки используются чаще других элементов пользовательского интерфейса, то второе место по популярности безусловно занимают флажки. В Swing эти элементы графического интерфейса реализуются с помощью объекта типа JCheckBox. Класс JCheckBox является производным от классов AbstractButton и JToggleButton. Следовательно, флажок — это особая разновидность кнопки.

В классе JCheckBox определен ряд конструкторов. Один из них имеет следующий вид:

```
JCheckBox(String строка)
```

Этот конструктор создает флажок с пояснительной надписью в виде строки, передаваемой через параметр *строка*.

При установке или сбросе флажка генерируется событие элемента. События элементов представляются классом ItemEvent и обрабатываются классами, реализующими интерфейс ItemListener. В этом интерфейсе определен только один метод, itemStateChanged(), общая форма объявления которого приведена ниже.

```
void itemStateChanged(ItemEvent ie)
```

Метод получает событие в параметре *ie.*

Для того чтобы получить ссылку на элемент, состояние которого изменилось, следует вызвать метод getItem() для объекта ItemEvent. Ниже приведена общая форма объявления этого метода.

```
Object getItem()
```

Возвращаемая этим методом ссылка должна быть приведена к типу обрабатываемого компонента, в данном случае — к типу JCheckBox.

Ассоциированный с флажком текст можно получить, вызвав метод getText(). Чтобы задать текст пояснительной надписи после создания флажка, следует вызвать метод setText(). Эти методы действуют точно так же, как и одноименные методы рассмотренного ранее класса JButton.

Самый простой способ определить состояние флажка — вызвать метод isSelected(), который объявляется следующим образом:

```
boolean isSelected()
```

Этот метод возвращает значение true, если флажок установлен, иначе — значение false.

Ниже приведен пример программы, демонстрирующий работу с флажками. В программе создаются три флажка: Альфа, Бета и Гамма. Всякий раз, когда состояние флажка изменяется, в окне программы появляются сведения о выполненном действии, а также перечисляются те флажки, которые установлены в данный момент. Результат выполнения данной программы приведен на рис. 16.4.

```
// Демонстрация использования флажков

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class CBDemo implements ItemListener {

   JLabel jlabSelected;
   JLabel jlabChanged;
   JCheckBox jcbAlpha;
   JCheckBox jcbBeta;
   JCheckBox jcbGamma;
```

```
CBDemo() {
   // Создать новый контейнер JFrame
   JFrame jfrm = new JFrame("Демонстрация флажков");

   // Задать объект FlowLayout для менеджера компоновки
   jfrm.setLayout(new FlowLayout());

   // Задать исходные размеры фрейма
   jfrm.setSize(280, 120);

   // Прекратить работу программы, если
   // пользователь закрывает приложение
   jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   // Создать пустые метки
   jlabSelected = new JLabel("");
   jlabChanged = new JLabel("");

   // Создать флажки
   jcbAlpha = new JCheckBox("Альфа"); ◄──┐
   jcbBeta = new JCheckBox("Бета");       │ Создание флажков
   jcbGamma = new JCheckBox("Гамма"); ◄──┘

   // События, генерируемые флажками, обрабатываются одним
   // методом itemStateChanged(), реализованным в классе CBDemo
   jcbAlpha.addItemListener(this);
   jcbBeta.addItemListener(this);
   jcbGamma.addItemListener(this);

   // Добавить флажки и метки на панель содержимого
   jfrm.add(jcbAlpha);
   jfrm.add(jcbBeta);
   jfrm.add(jcbGamma);
   jfrm.add(jlabChanged);
   jfrm.add(jlabSelected);

   // Отобразить фрейм
   jfrm.setVisible(true);
}

// Обработчик для флажков
public void itemStateChanged(ItemEvent ie) { ◄── Обработка событий флажков
   String str = "";

   // Получить ссылку на флажок, с которым связано событие
   JCheckBox cb = (JCheckBox) ie.getItem(); ◄── Получение ссылки
                                                 на измененный флажок

   // Сообщить о том, состояние какого флажка изменилось
   if(cb.isSelected()) ◄── Определить, что произошло
      jlabChanged.setText(cb.getText() + " был выбран.");
   else
      jlabChanged.setText(cb.getText() + " был сброшен.");
```

```
        // Сообщить обо всех установленных флажках
        if(jcbAlpha.isSelected()) {
            str += "Альфа ";
        }
        if(jcbBeta.isSelected()) {
            str += "Бета ";
        }
        if(jcbGamma.isSelected()) {
            str += "Гамма";
        }

        jlabSelected.setText("Выбраны флажки: " + str);
    }

    public static void main(String args[]) {
        // Создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new CBDemo();
            }
        });
    }
}
```

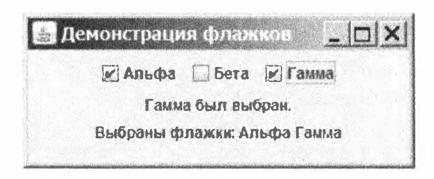


Рис. 16.4. Результат выполнения программы `CBDemo`

Наибольший интерес в рассматриваемом примере представляет метод `itemStateChanged()`, предназначенный для обработки событий элементов (в данном случае флажков). Он решает две задачи: сообщает, установлен или сброшен флажок, и отображает список всех установленных флажков. Сначала определяется ссылка на компонент, сгенерировавший событие `ItemEvent`. Это происходит в следующей строке кода:

```
JCheckBox cb = (JCheckBox) ie.getItem();
```

Приведение к типу `JCheckBox` необходимо потому, что метод `getItem()` возвращает ссылку на объект типа `Object`. Далее метод `itemStateChanged()` вызывает метод `isSelected()` для объекта `cb`, чтобы определить текущее состояние флажка. Если метод `isSelected()` возвращает значение `true`, значит, флажок установлен, а если `false` — флажок сброшен. Затем с помощью метки `jlabChanged` отображаются сведения о выполненном действии.

И наконец, метод `itemStateChanged()` проверяет состояние каждого флажка и формирует строку с именами установленных флажков. Эта строка отображается в окне программы с помощью метки `jlabSelected`.

Класс Jlist

Класс JList — это базовый класс Swing, обеспечивающий поддержку списков с возможностью выбора одного или нескольких элементов. Чаще всего в списках содержатся строки, но можно создавать списки, элементами которых являются любые отображаемые объекты. Сфера применимости класса JList настолько широкая, что вам несомненно уже приходилось с ним сталкиваться.

Ранее элементы JList предоставлялись в виде ссылок на объекты типа Object. Однако уже в версии JDK 7 компонент JList стал обобщенным и теперь объявляется так:

```
class JList<E>
```

где E обозначает тип элементов списка. Таким образом, теперь класс JList является безопасным в отношении контроля типов.

Класс JList имеет несколько конструкторов. Далее мы будем использовать конструктор следующего вида:

```
JList(E[] элементы)
```

Этот конструктор создает список JList, содержащий элементы в виде массива, указанного с помощью параметра *элементы*.

Несмотря на то что с классом JList можно работать напрямую, в большинстве случаев его помещают в оболочку класса JScrollPane, автоматически поддерживающего прокрутку содержимого. Вот как выглядит конструктор этого класса:

```
JScrollPane(Component компонент)
```

где *компонент* — это конкретный компонент, нуждающийся в возможностях прокрутки (в данном случае это компонент типа JList). Помещение компонента JList в контейнер JScrollPane автоматически обеспечивает возможность прокрутки длинных списков. Это не только упрощает построение графического пользовательского интерфейса, но и позволяет варьировать количество элементов списка, не изменяя при этом размеры самого компонента JList.

При выборе пользователем элемента компонент JList генерирует событие ListSelectionEvent. Это же событие генерируется и при отмене выбора элемента. Для его обработки используется реализация интерфейса ListSelectionListener, входящего в пакет javax.swing.event. В этом слушателе событий определен только один метод:

```
void valueChanged(ListSelectionEvent le)
```

где *le* обозначает ссылку на объект, сгенерировавший событие. И хотя в классе ListSelectionEvent имеются собственные методы, позволяющие следить за состоянием списка, вы будете чаще использовать для этой цели непосредственно сам объект JList. Класс ListSelectionEvent также входит в пакет javax.swing.event.

По умолчанию класс `JList` позволяет выбирать несколько диапазонов элементов в списке, но это поведение можно изменить, вызвав метод `setSelectionMode()`, определенный в классе `JList`.

```
void setSelectionMode(int режим)
```

где параметр *режим* задает режим выбора элементов. Значение этого параметра должно совпадать с одной из приведенных ниже констант, определенных в интерфейсе `ListSelectionModel`, входящем в пакет `javax.swing`:

```
SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION
```

По умолчанию устанавливается режим `MULTIPLE_INTERVAL_SELECTION`, позволяющий выбирать несколько диапазонов элементов в списке. В режиме `SINGLE_INTERVAL_SELECTION` разрешен выбор только одного диапазона элементов, в режиме `SINGLE_SELECTION` — только одного элемента. Очевидно, что выбор единственного элемента возможен и в остальных двух режимах, предназначенных для группового выбора элементов.

Индекс первого выбранного элемента (в режиме `SINGLE_SELECTION`) можно получить, вызвав метод `getSelectedIndex()`, синтаксис объявления которого представлен ниже.

```
int getSelectedIndex()
```

Индексирование начинается с нуля. Поэтому, если выбран первый элемент, метод вернет значение 0. Если же ни один из элементов не выбран, возвращается значение -1.

Для получения массива, содержащего все выбранные элементы, следует вызвать метод `getSelectedIndices()`.

```
int[] getSelectedIndices()
```

В возвращаемом массиве индексы упорядочены от меньшего к большему. Если возвращается массив нулевой длины, то это означает, что ни один из элементов не выбран.

Ниже приведен пример программы, в которой демонстрируется использование простого списка `JList`, содержащего имена. Всякий раз, когда пользователь выбирает имя из списка, генерируется событие `ListSelectionEvent`, которое обрабатывается методом `valueChanged()`, объявленным в интерфейсе `ListSelectionListener`. Этот метод определяет индекс выбранного элемента и отображает соответствующее имя. Результат выполнения программы приведен на рис. 16.5.

```
// Демонстрация использования простого списка JList

import javax.swing.*;
import javax.swing.event.*;
```

```
import java.awt.*;
import java.awt.event.*;

class ListDemo implements ListSelectionListener {

   JList<String> jlst;
   JLabel jlab;
   JScrollPane jscrlp;

   // Создать массив имен
   String names[] = { "Мария", "Иван", "Светлана",
                      "Александр", "Евгения", "Наталья",
                      "Аркадий", "Валентина", "Борис",
                      "Андрей", "Степан", "Владислав" };   // Это содержимое списка JList

   ListDemo() {
      // Создать новый контейнер JFrame
      JFrame jfrm = new JFrame("Демонстрация списка");

      // Задать объект FlowLayout для менеджера компоновки
      jfrm.setLayout(new FlowLayout());

      // Задать исходные размеры фрейма
      jfrm.setSize(200, 160);

      // Прекратить работу программу, если
      // пользователь закрывает приложение
      jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Создать объект JList
      jlst = new JList<String>(names);  ◄── Создание списка

      // Задать режим выбора одиночных элементов
      jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);  ◄── Переключение в режим выбора одиночных элементов

      // Добавить список на панель с полосами прокрутки
      jscrlp = new JScrollPane(jlst);  ◄── Добавление списка на панель с полосами прокрутки

      // Задать предпочтительные размеры прокручиваемой панели
      jscrlp.setPreferredSize(new Dimension(120, 90));

      // Создать метку для отображения результатов выбора
      jlab = new JLabel("Выберите имя");

      // Добавить обработчик для событий списка
      jlst.addListSelectionListener(this);  ◄── Прослушивание событий, связанных с выбором элементов списка

      // Добавить список и метку на панель содержимого
      jfrm.add(jscrlp);
      jfrm.add(jlab);
```

```
        // Отобразить фрейм
        jfrm.setVisible(true);
    }
                                                    Обработка событий списка
    // Обработка событий списка
    public void valueChanged(ListSelectionEvent le) { ◄
        // Получить индекс элемента, состояние выбора
        // которого было изменено
        int idx = jlst.getSelectedIndex(); ◄— Получение индекса выделенных/
                                              не выделенных элементов

        // Отобразить результат выбора, если элемент был выбран
        if(idx != -1)
            jlab.setText("Текущее выделение: " + names[idx]);
        else // иначе еще раз предложить сделать выбор
            jlab.setText("Выберите имя");
    }

    public static void main(String args[]) {
         // Создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new ListDemo();
            }
        });
    }
}
```

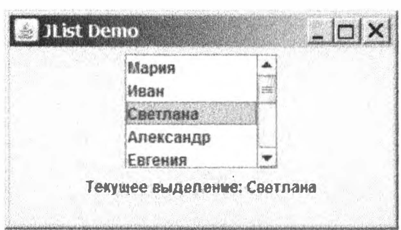


Рис. 16.5. Результат выполнения программы `JListDemo`

Рассмотрим исходный код программы более подробно. Прежде всего, обратите внимание на объявление массива `names` в начале программы. Он инициализируется строками, содержащими различные имена. В конструкторе `ListDemo()` массив `names` используется для создания объекта `jlst`. Конструктор, которому в качестве параметра передается массив, как это имеет место в данном случае, автоматически создает экземпляр класса `JList`, содержащий элементы массива. Следовательно, формируемый список будет состоять из имен, хранящихся в массиве `names`.

Далее устанавливается режим, допускающий выбор только одного элемента из списка. Затем объект `jlst` помещается в контейнер `JScrollPane`, а для панели прокрутки задаются предпочтительные размеры 120×90. Это делается ради компактности и удобства использования данного компонента. Для задания

предпочтительных размеров компонента служит метод `setPreferredSize()`. Как правило, предпочтительные размеры определяют фактические размеры компонента, но не следует забывать, что некоторые менеджеры компоновки могут игнорировать подобные запросы на установку размеров компонентов.

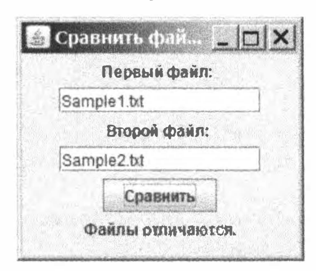
Когда пользователь выбирает элемент или изменяет свой выбор, генерируется событие выбора элемента. Для получения индекса выбранного элемента в обработчике `valueChanged()` вызывается метод `getSelectedIndex()`. Поскольку для списка был задан режим, ограничивающий выбор только одним элементом, этот индекс однозначно определяет выбранный элемент. Затем индекс используется для обращения к массиву `names` и получения имени выбранного элемента. Обратите внимание на то, что в данной программе проверяется, равен ли индекс значению –1. Вспомните, что это значение возвращается в том случае, если не был выбран ни один элемент. Именно это происходит в ситуациях, когда событие генерируется в результате отмены пользователем своего выбора. Не забывайте: событие выбора элемента генерируется как при выборе элемента, так и при отмене выбора.

Упражнение 16.1

Утилита сравнения файлов, созданная на основе Swing

`SwingFC.java`

Несмотря на то что вы ознакомились лишь с небольшой частью библиотеки Swing, вы уже в состоянии создавать приложения, имеющие практическую ценность. В упражнении 10.1 была создана консольная утилита сравнения файлов. Теперь нам предстоит снабдить ее пользовательским интерфейсом, созданным на основе компонентов Swing. Это позволит значительно улучшить внешний вид утилиты и сделать ее более удобной в использовании. Ниже показано, как выглядит рабочее окно утилиты сравнения файлов, созданной на основе Swing.



В процессе работы над данным проектом вы сможете сами убедиться, насколько библиотека Swing упрощает создание приложений с графическим пользовательским интерфейсом. Поэтапное описание процесса создания программы приведено ниже.

1. Создайте файл `SwingFC.java` и введите приведенные ниже комментарии и инструкции `import`.

```
/*
   Упражнение 16.1.

   Утилита сравнения файлов на основе Swing.

*/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
```

2. Создайте класс `SwingFC`, начав с приведенного ниже исходного кода.

```
public class SwingFC implements ActionListener {

   JTextField jtfFirst; // хранит имя первого файла
   JTextField jtfSecond; // хранит имя второго файла

   JButton jbtnComp; // кнопка для запуска операции сравнения файлов

   JLabel jlabFirst, jlabSecond; // метки, отображающие
                                 // подсказки для пользователя
   JLabel jlabResult; // метка для отображения результата
                      // сравнения и сообщений об ошибках
```

   Имена сравниваемых файлов указываются в текстовых полях `jtfFirst` и `jtfSecond`. Для того чтобы начать сравнение файлов, указанных в этих полях, пользователь должен щелкнуть на кнопке `jbtnComp`. Подсказки для пользователя отображаются с помощью меток `jlabFirst` и `jlabSecond`. Результаты сравнения и сообщения об ошибках отображаются с помощью метки `jlabResult`.

3. Создайте конструктор класса `SwingFC`.

```
SwingFC() {

   // Создать новый контейнер JFrame
   JFrame jfrm = new JFrame("Сравнить файлы");

   // Задать объект FlowLayout для менеджера компоновки
   jfrm.setLayout(new FlowLayout());

   // Задать исходные размеры фрейма
   jfrm.setSize(200, 190);

   // Прекратить работу программы, если
   // пользователь закрывает приложение
   jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    // Создать поля для ввода имен файлов
    jtfFirst = new JTextField(14);
    jtfSecond = new JTextField(14);

    // Задать команды действия для текстовых полей
    jtfFirst.setActionCommand("ФайлА");
    jtfSecond.setActionCommand("ФайлБ");

    // Создать кнопку сравнения
    JButton jbtnComp = new JButton("Сравнить");

    // Добавить слушателя событий для кнопки
    jbtnComp.addActionListener(this);

    // Создать метки
    jlabFirst = new JLabel("Первый файл: ");
    jlabSecond = new JLabel("Второй файл: ");
    jlabResult = new JLabel("");

    // Добавить компоненты на панель содержимого
    jfrm.add(jlabFirst);
    jfrm.add(jtfFirst);
    jfrm.add(jlabSecond);
    jfrm.add(jtfSecond);
    jfrm.add(jbtnComp);
    jfrm.add(jlabResult);

    // Отобразить фрейм
    jfrm.setVisible(true);
}
```

Большая часть исходного кода этого конструктора должна быть вам уже знакома. Обратите лишь внимание на следующую особенность: слушатель событий действий задается только для кнопки `jbtnComp`, в то время как аналогичные слушатели для текстовых полей не добавляются. Дело в том, что содержимое полей ввода текста требуется только в тот момент, когда нажимается кнопка **Сравнить**, а все остальное время необходимости в знании их содержимого не возникает. Поэтому реагировать на события, связанные с текстовыми полями, нет никакого смысла. Когда вы начнете писать реальные программы с использованием библиотеки Swing, вы обнаружите, что подобная ситуация с текстовыми полями является довольно распространенной.

4. Начните создавать обработчик событий `actionPerformed()`, как показано ниже. Этот метод будет вызываться после щелчка на кнопке **Сравнить**.

```
// Сравнить файлы после щелчка на кнопке
public void actionPerformed(ActionEvent ae) {
    int i=0, j=0;
```

```
// Сначала убедиться в том, что введены имена обоих файлов
if(jtfFirst.getText().equals("")) {
   jlabResult.setText("Отсутствует имя первого файла.");
   return;
}
if(jtfSecond.getText().equals("")) {
   jlabResult.setText("Отсутствует имя второго файла.");
   return;
}
```

Здесь проверяется, ввел ли пользователь имена обоих файлов в соответствующих текстовых полях. Если какое-то из этих полей осталось пустым, выводится соответствующее сообщение и обработчик завершает работу.

5. Завершите создание обработчика событий, введя приведенный ниже код, который открывает файлы и сравнивает их содержимое.

```
   // Сравнить файлы, используя инструкцию try с ресурсами
   try (FileInputStream f1 = new FileInputStream(jtfFirst.getText());
        FileInputStream f2 = new FileInputStream(jtfSecond.getText()))
   {
      // Сравнить содержимое обоих файлов
      do {
         i = f1.read();
         j = f2.read();
         if(i != j) break;
      } while(i != -1 && j != -1);

      if(i != j)
         jlabResult.setText("Файлы отличаются.");
      else
         jlabResult.setText("Файлы одинаковы.");
   } catch(IOException exc) {
      jlabResult.setText("Ошибка файла");
   }
}
```

6. Добавьте в класс `SwingFC` метод `main()`.

```
   public static void main(String args[]) {
      // Создать фрейм в потоке диспетчеризации событий
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            new SwingFC();
         }
      });
   }
}
```

7. Ниже приведен полный исходный код утилиты сравнения файлов.

```
/*
   Упражнение 16.1.

   Утилита сравнения файлов на основе Swing.
```

```
*/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

public class SwingFC implements ActionListener {

   JTextField jtfFirst; // хранит имя первого файла
   JTextField jtfSecond; // хранит имя второго файла

   JButton jbtnComp; // кнопка для запуска операции сравнения файлов

   JLabel jlabFirst, jlabSecond; // метки, отображающие
                                 // подсказки для пользователя
   JLabel jlabResult; // метка для отображения результата
                      // сравнения и сообщений об ошибках

   SwingFC() {

      // Создать новый контейнер JFrame
      JFrame jfrm = new JFrame("Сравнить файлы");

      // Задать объект FlowLayout для менеджера компоновки
      jfrm.setLayout(new FlowLayout());

      // Задать исходные размеры фрейма
      jfrm.setSize(200, 190);

      // Прекратить работу программы, если
      // пользователь закрывает приложение
      jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Создать поля для ввода имен файлов
      jtfFirst = new JTextField(14);
      jtfSecond = new JTextField(14);

      // Задать команды действия для текстовых полей
      jtfFirst.setActionCommand("ФайлА");
      jtfSecond.setActionCommand("ФайлБ");

      // Создать кнопку сравнения
      JButton jbtnComp = new JButton("Сравнить");

      // Добавить слушатель событий для кнопки
      jbtnComp.addActionListener(this);

      // Создать метки
      jlabFirst = new JLabel("Первый файл: ");
      jlabSecond = new JLabel("Второй файл: ");
      jlabResult = new JLabel("");
```

```
        // Добавить компоненты на панель содержимого
        jfrm.add(jlabFirst);
        jfrm.add(jtfFirst);
        jfrm.add(jlabSecond);
        jfrm.add(jtfSecond);
        jfrm.add(jbtnComp);
        jfrm.add(jlabResult);

        // Отобразить фрейм
        jfrm.setVisible(true);
    }

    // Сравнить файлы после щелчка на кнопке
    public void actionPerformed(ActionEvent ae) {
        int i=0, j=0;

        // Сначала убедиться в том, что введены имена обоих файлов
        if(jtfFirst.getText().equals("")) {
            jlabResult.setText("Отсутствует имя первого файла.");
            return;
        }
        if(jtfSecond.getText().equals("")) {
            jlabResult.setText("Отсутствует имя второго файла.");
            return;
        }

        // Сравнить файлы, используя инструкцию try с ресурсами
        try (FileInputStream f1 = new
                           FileInputStream(jtfFirst.getText());
             FileInputStream f2 = new
                           FileInputStream(jtfSecond.getText()))
        {
            // Сравнить содержимое обоих файлов
            do {
                i = f1.read();
                j = f2.read();
                if(i != j) break;
            } while(i != -1 && j != -1);

            if(i != j)
                jlabResult.setText("Файлы отличаются.");
            else
                jlabResult.setText("Файлы одинаковы.");
        } catch(IOException exc) {
                jlabResult.setText("Ошибка файла");
        }
    }

    public static void main(String args[]) {
        // Создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
```

```
            public void run() {
               new SwingFC();
            }
         });
      }
   }
```

Применение анонимных внутренних классов или лямбда-выражений для обработки событий

В рассмотренных ранее примерах применялся достаточно простой подход к обработке событий, когда основной класс приложения реализовывал интерфейс соответствующего слушателя событий, а все события передавались для обработки экземпляру этого класса. И хотя такой подход вполне пригоден для написания приложений с пользовательским интерфейсом, его нельзя рассматривать как единственно возможный. В подобных программах могут применяться и другие способы обработки событий, происходящих в пользовательском интерфейсе. Например, для каждого события можно реализовать слушатель в отдельном классе. Благодаря этому разнородные события будут обрабатываться в разных классах, и эти классы будут отделены от основного класса приложения. Но есть два более мощных подхода. Во-первых, слушатели событий можно реализовать с помощью *анонимных внутренних классов*.Во-вторых, иногда обработку событий можно организовать с помощью лямбда-выражений. Рассмотрим оба подхода.

У анонимного внутреннего класса нет имени, а экземпляр такого класса получают динамически по мере необходимости. Анонимные внутренние классы позволяют значительно упростить создание обработчиков для некоторых видов событий. Допустим, имеется компонент `jbtn` типа `JButton`. Тогда слушатель событий кнопки может быть реализован следующим образом.

```
jbtn.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent ae) {
      // обработка события
   }
});
```

В данном примере используется анонимный внутренний класс, реализующий интерфейс `ActionListener`. Обратите внимание на синтаксис, используемый при создании этого класса. Тело внутреннего класса начинается после символа {, следующего за выражением `new ActionListener()`. Обратите также внимание на то, что вызов метода `addActionListener()` завершается закрывающей скобкой и точкой с запятой, т.е. как обычно. Такой синтаксис применяется при создании анонимных внутренних классов, предназначенных для обработки любых событий. Естсественно, для разнородных событий задаются разные слушатели и реализуются разные методы.

Преимущество анонимного внутреннего класса заключается, в частности, в том, что компонент, вызывающий методы этого класса, заранее известен. Так, в предыдущем примере не было никакой необходимости вызывать метод `getActionCommand()`, чтобы выяснить, каким именно компонентом было сгенерировано событие, поскольку метод `actionPerformed()` может быть вызван в подобной реализации только при наступлении событий, сгенерированных компонентом `jbtn`.

В случае событий, слушатели которых реализуют функциональный интерфейс, обработка событий может быть выполнена с использованием лямбда-выражений. Так, лямбда-выражения могут использоваться для обработки событий действий, поскольку в интерфейсе `ActionListener` определен только один абстрактный метод — `actionPerformed()`. Реализация интерфейса `ActionListener` с помощью лямбда-выражений — это более компактная альтернатива явному объявлению анонимного внутреннего класса. Например, для компонента `jbtn` типа `JButton` слушатель событий может быть реализован следующим образом.

```
jbtn.addActionListener((ae) -> {
   // обработка события
});
```

Как и в случае подхода, в котором используется анонимный внутренний класс, здесь известен объект, генерирующий событие. В данном случае лямбда-выражение применяется только к кнопке `jbtn`.

Очевидно, что в ситуациях, когда событие может быть обработано с помощью одиночного лямбда-выражения, в использовании блочных лямбда-выражений нет никакой необходимости. Ниже в качестве примера представлен обработчик событий действий для нажатия кнопки в рассмотренной ранее программе `ButtonDemo`. В нем требуется только одиночное лямбда-выражение.

```
jbtnUp.addActionListener((ae) -> jlab.setText("Кнопка отпущена"));
```

Заметьте, насколько короче стал код по сравнению с его первоначальным вариантом. Он также короче того кода, который вы получили бы, если бы использовали анонимный внутренний класс.

Если говорить в общем, то лямбда-выражения могут использоваться для обработки событий во всех случаях, когда слушатель объявляет функциональный интерфейс. Например, `ItemListener` является функциональным интерфейсом. Разумеется, выбор того, следует ли использовать традиционный подход, анонимные внутренние классы или лямбда-выражения, определяется спецификой вашего приложения.

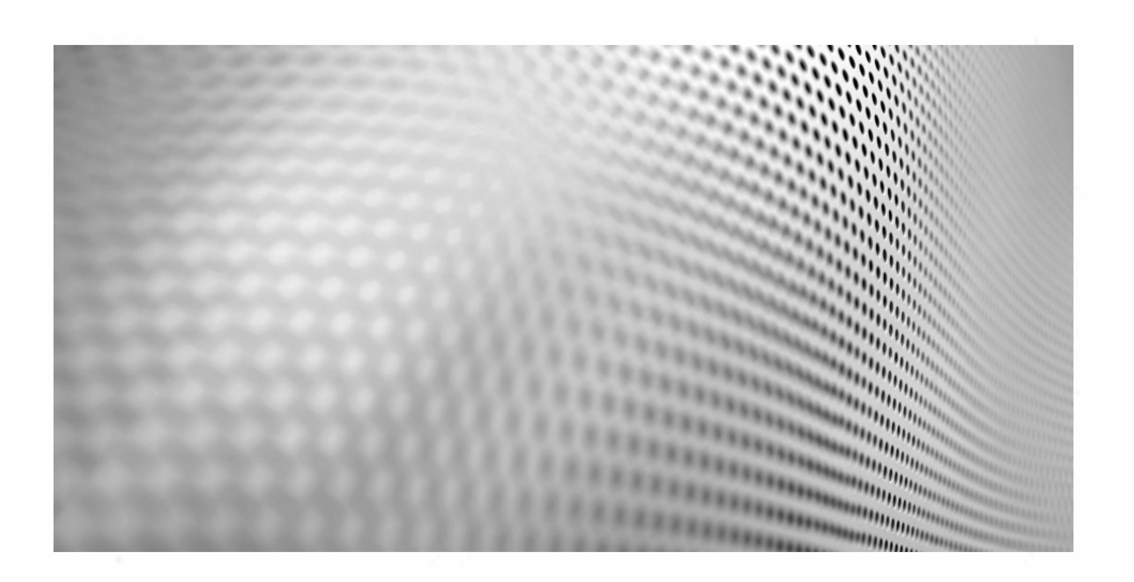
Вопросы и упражнения для самопроверки

1. Компоненты AWT являются тяжеловесными, а компоненты Swing — ________________.
2. Можно ли изменить стиль оформления компонента Swing? Если да, то какое средство позволяет это сделать?
3. Какой контейнер верхнего уровня чаще всего используется в приложениях?
4. Контейнер верхнего уровня содержит несколько панелей. На какой из них располагаются компоненты?
5. Как создать метку, отображающую сообщение `"Выберите элемент списка"`?
6. В каком потоке должно осуществляться любое взаимодействие с компонентами графического пользовательского интерфейса?
7. Какая команда действия связывается по умолчанию с компонентом `JButton`? Как изменить команду действия?
8. Какое событие генерируется при щелчке на кнопке?
9. Как создать текстовое поле шириной 32 столбца?
10. Можно ли задать команду действия для компонента `JTextField`? Если можно, то как это сделать?
11. С помощью какого компонента Swing можно создать флажок? Какое событие генерируется при установке или сбросе флажка?
12. Компонент `JList` отображает список элементов, которые может выбирать пользователь. Верно или неверно?
13. Какое событие генерируется при выборе или отмене выбора элемента из списка типа `JList`?
14. В каком методе задается режим выбора элементов списка `JList`? С помощью какого метода можно получить индекс первого выбранного элемента?
15. Добавьте в утилиту сравнения файлов, созданную в упражнении 16.1, флажок со следующей пояснительной надписью: `Показывать позицию расхождения`. Если этот флажок установлен, программа должна отображать позицию, в которой обнаружено первое расхождение в содержимом сравниваемых файлов.
16. Измените программу `ListDemo` таким образом, чтобы она допускала выбор нескольких элементов списка.
17. Задание повышенной сложности. Преобразуйте класс `Help`, созданный в упражнении 4.1, в программу Swing с графическим пользовательским интерфейсом. Сведения о ключевых словах (`for`, `while`, `switch` и др.)

должны отображаться с помощью компонента JList. При выборе пользователем элемента списка должно выводиться описание синтаксиса выбранного ключевого слова. Для отображения многострочного текста внутри метки можно использовать средства HTML. В этом случае текст должен начинаться с дескриптора <html> и завершаться дескриптором </html>. В итоге текст будет автоматически размечен в виде HTML-документа. Помимо прочих преимуществ, такая разметка текста позволяет создавать многострочные метки. В качестве примера ниже приведена строка кода, в которой создается метка, отображающая две текстовые строки: первой выводится строка "Top", а под ней — строка "Bottom".

```
JLabel jlabhtml = new JLabel("<html>Top<br>Bottom</html>");
```

Решение этого упражнения не приводится. Ведь уровень вашей подготовки уже настолько высок, что вы в состоянии самостоятельно разрабатывать программы на Java!



Глава 17

Введение в JavaFX

В этой главе...

- Основные понятия JavaFX: платформа, сцена, узел, граф сцены
- Жизненный цикл приложений JavaFX
- Общая форма приложения JavaFX
- Запуск приложения JavaFX
- Создание компонента `Label`
- Использование компонента `Button`
- Обработка событий
- Использование компонента `CheckBox`
- Работа с компонентом `ListView`
- Создание компонента `TextField`
- Добавление эффектов
- Применение преобразований

В стремительно развивающемся компьютерном мире постоянны лишь изменения, а наука и искусство программирования непрерывно развиваются, осваивая все новые рубежи. Поэтому нет ничего удивительного в том, что библиотеки Java, поддерживающие графический интерфейс пользователя (GUI), также оказались вовлеченными в этот процесс. Напомним, что в Java первой такой библиотекой была AWT. Вслед за AWT была разработана библиотека Swing, которая намного превосходила по возможностям свою предшественницу. Несмотря на успешность библиотеки Swing, создавать с ее помощью всевозможные визуальные эффекты, столь востребованные во многих современных приложениях, довольно затруднительно. Кроме того, изменения коснулись самих концептуальных основ проектирования пользовательских интерфейсов, что заставляло искать новые подходы к разработке GUI. Ответом разработчиков на запросы Java-сообщества стала библиотека JavaFX, представляющая собой GUI-фреймворк следующего поколения. Данная глава содержит минимальный набор необходимых сведений, знание которых позволит вам в кратчайшие сроки приступить к работе с этой новой мощной системой.

Важно отметить, что развитие библиотеки JavaFX происходило в два этапа. Ранние версии JavaFX базировались на языке сценариев *JavaFX Script*. Однако в более поздних версиях, начиная с JavaFX 2.0, этот язык уже не поддерживается, и вместо него предлагается новый программный интерфейс для создания

JavaFX-приложений полностью на языке Java. В JDK 7 (обновление 4) и более поздних версиях Java библиотека JavaFX входит в стандартный комплект поставки. Последней версией библиотеки, включенной в комплект JDK 10, является JavaFX 10. Далее рассматривается JavaFX 10 как самая последняя версия библиотеки JavaFX на момент написания книги. Соответственно, в тех местах, где термин JavaFX встречается без конкретизации номера версии, под ним подразумевается версия JavaFX 10.

Прежде чем продолжить, следует дать ответ на один вопрос, который естественным образом возникает в отношении JavaFX: предназначалась ли библиотека JavaFX для того, чтобы заменить собой Swing? По сути, так оно и было. Однако некоторое время Swing еще будет оставаться неотъемлемой частью программирования на языке Java. Это обусловлено наличием больших объемов унаследованного кода с графическим интерфейсом на основе Swing. Немаловажен и тот факт, что в настоящее время огромное количество программистов, освоивших технологию Swing, продолжают использовать ее в своих разработках. Тем не менее абсолютно очевидно, что будущее принадлежит JavaFX. Иными словами, любой программист, пишущий программы на Java, должен владеть технологией JavaFX.

Примечание

В данной главе предполагается, что вы уже имеете представление о том, что такое графический интерфейс пользователя и как обрабатываются события (см. главу 16).

Базовые понятия JavaFX

Прежде чем приступить к созданию приложения JavaFX, вам необходимо ознакомиться с основными понятиями и возможностями этой технологии. Несмотря на некоторое сходство JavaFX с другими графическими интерфейсами Java, такими как AWT и Swing, между ними имеются существенные различия. Аналогично Swing, компоненты JavaFX относятся к категории легковесных, а способы обработки событий просты и интуитивно понятны. Но если говорить об общих принципах организации библиотеки и взаимосвязи ее основных компонентов, то JavaFX значительно отличается как от Swing, так и от AWT. Поэтому вам стоит внимательно изучить материал, изложенный в следующих разделах.

Пакеты JavaFX

Библиотека JavaFX содержится в пакетах, имена которых начинаются с префикса `javafx`. На момент написания данной книги JavaFX API включал более 30 пакетов. В качестве примера назовем четыре: `javafx.application`, `javafx.stage`, `javafx.scene` и `javafx.scene.layout`. В этой главе нам

понадобится лишь несколько пакетов JavaFX, однако вам стоит потратить какое-то время на краткое ознакомление с остальными пакетами библиотеки, поскольку спектр ее возможностей очень обширен.

Классы Stage и Scene

В качестве центральной метафоры, на основе которой создавалась библиотека JavaFX, разработчики выбрали *театральные подмостки* (stage). Как и в любом реальном театре, подмостки служат сценической площадкой, на которой разыгрываются *сцены* (scenes). Образно говоря, подмостки, или театральная платформа, определяют пространственные границы для сцен, которые, в свою очередь, формируются из других элементов. Аналогично этому любое JavaFX-приложение содержит по крайней мере одну платформу и одну сцену. В JavaFX API эти элементы инкапсулируются классами `Stage` и `Scene`. Чтобы создать JavaFX-приложение, нужно добавить в объект `Stage` хотя бы один объект `Scene`. Рассмотрим более детально, что собой представляют два этих класса.

Класс `Stage` — это контейнер верхнего уровня. Все приложения JavaFX автоматически получают доступ к одному контейнеру класса `Stage`, называемому *основной платформой* (primary stage). Основная платформа предоставляется исполняющей средой при запуске приложения. Несмотря на возможность создания нескольких платформ, в большинстве случаев одной платформы оказывается достаточно.

Как уже отмечалось, класс `Scene` — это контейнер для элементов, составляющих сцену. Этими элементами могут быть кнопки и флажки, текст и графика. Для создания сцены вы будете добавлять эти элементы в экземпляр класса `Scene`.

Узлы и графы сцены

Отдельные элементы сцены называют *узлами* (nodes). Например, узлом является кнопка. В то же время узлы сами по себе могут состоять из групп узлов. Кроме того, у любого узла могут быть дочерние узлы. Узел, имеющий дочерние узлы, называется *родительским* (parent node), или *узлом ветвления* (branch node). Узлы, не имеющие дочерних узлов, являются оконечными и называются *листьями* (leaves). Совокупность всех узлов сцены называется *графом сцены* (scene graph) и образует *дерево* (tree), т.е. иерархическую структуру узлов.

Особую роль в графе сцены играет корневой узел, или *корень* (root). Им является узел верхнего уровня, и это единственный узел в графе сцены, не имеющий родительского узла. Таким образом, за исключением корневого узла, все остальные узлы имеют родителей и являются непосредственными или косвенными потомками корневого узла.

Класс `Node` является базовым для всех типов узлов. Существуют также другие классы, являющиеся прямыми или косвенными потомками класса `Node`. В частности, это классы `Parent`, `Group`, `Region` и `Control`.

Панели компоновки

Библиотека JavaFX предоставляет несколько панелей компоновки, с помощью которых можно управлять процессом размещения элементов в сцене. Например, класс FlowPane обеспечивает плавающую компоновку, а класс GridPane — табличную компоновку элементов в виде строк и столбцов. Также доступен ряд других менеджеров компоновки, например BorderPane (аналогичен компоновщику BorderLayout библиотеки AWT). Соответствующие классы находятся в пакете javafx.scene.layout.

Класс Application и жизненный цикл приложения

Приложение JavaFX должно быть подклассом класса Application, находящегося в пакете javafx.application. Таким образом, класс приложения должен расширять класс Application. Класс Application определяет три метода, управляющих жизненным циклом приложения: init(), start() и stop(), которые приложение может переопределить. Синтаксис их объявлений представлен ниже в порядке вызова методов.

```
void init()

abstract void start(Stage основная_платформа)

void stop()
```

Метод init() вызывается в начале выполнения приложения и служит для инициализации всех необходимых переменных. Однако, как будет показано далее, его *нельзя* использовать для создания платформы или формирования сцены. Если никакая инициализация не требуется, то данный метод можно не переопределять, поскольку по умолчанию предоставляется его пустая версия.

Метод start() вызывается после метода init(). Именно с него начинается работа приложения, и его можно использовать для конструирования и установки параметров сцены. Обратите внимание на то, что в качестве аргумента ему передается ссылка на объект Stage. Этот объект и есть та самая основная платформа, которую предоставляет исполняющая среда. Заметьте также, что этот метод объявлен как абстрактный, и поэтому он должен переопределяться в приложении.

По завершении работы приложения вызывается метод stop(). Именно в нем выполняются все рутинные операции, связанные со сборкой мусора и освобождением ресурсов, захваченных приложением. Если такие действия не требуются, то метод можно не переопределять, поскольку по умолчанию предоставляется его пустая версия.

Запуск приложения JavaFX

Чтобы запустить автономное JavaFX-приложение, следует вызвать метод launch(), определенный в интерфейсе Application. Этот метод имеет две формы объявления. Ниже приведена та из них, которая используется в данной главе.

```
public static void launch(String ...аргументы)
```

Здесь параметр *аргументы* — это список строк (возможно, пустой), обычно являющихся аргументами командной строки. Вызов метода `launch()` приводит к загрузке приложения, сопровождающейся последующими вызовами методов `init()` и `start()`. Возврат из метода `launch()` происходит лишь тогда, когда приложение завершает работу. Данная версия метода `launch()` загружает приложение в класс, который расширяет класс `Application` и является точкой входа в приложение. Вторая форма метода `launch()` позволяет указать в качестве точки входа класс, отличный от того, в котором вызывается метод.

Прежде чем продолжить, необходимо сделать одно важное замечание: приложения, упакованные с помощью утилиты `javafxpackager` (или эквивалентного ему средства интегрированной среды разработки), не нуждаются в вызове метода `launch()`. Вместе с тем его включение в приложение во многих случаях упрощает процессы тестирования и отладки и позволяет использовать программу, не создавая JAR-файл. Поэтому в данной главе вызов метода `launch()` всегда включается в приложение.

Каркас приложения JavaFX

Все приложения JavaFX создаются на основе одного и того же базового каркаса. Поэтому, прежде чем использовать другие возможности, полезно изучить, что собой представляет такой каркас. Это позволит не только продемонстрировать общую структуру JavaFX-приложения, но и показать, как запускается приложение и вызываются методы жизненного цикла. Приложение будет выводить на консоль сообщения, подсказывающие, когда именно вызывается тот или иной метод. Код приложения показан ниже.

```
// Каркас приложения JavaFX

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

   public static void main(String[] args) {

      System.out.println("Запуск приложения JavaFX");

      // Запустить приложение JavaFX, вызвав метод launch()
      launch(args);
   }

   // Переопределить метод init()
   public void init() {
      System.out.println("В теле метода init()");
   }
```

```
    // Переопределить метод start()
    public void start(Stage myStage) {

        System.out.println("В теле метода start()");

        // Задать заголовок окна приложения
        myStage.setTitle("Каркас приложения JavaFX");

        // Создать корневой узел. В данном случае
        // используется плавающая компоновка, но возможны
        // и другие варианты.
        FlowPane rootNode = new FlowPane(); ◄——— Создание корневого узла

        // Создать сцену
        Scene myScene = new Scene(rootNode, 300, 200); ◄——— Создание сцены

        // Установить сцену на платформе
        myStage.setScene(myScene); ◄——— Установка сцены на платформе

        // Отобразить платформу вместе с ее сценой
        myStage.show(); ◄——— Отображение сцены
    }

    // Переопределить метод stop()
    public void stop() {
        System.out.println("В теле метода stop()");
    }
}
```

Конечно, это совсем небольшое приложение, но его можно скомпилировать и выполнить. В итоге мы получаем пустое окно, однако при этом на консоль выводится следующий результат.

```
Запуск приложения JavaFX
В теле метода init()
В теле метода start()
```

При закрытии окна на консоли отображается следующее сообщение:

```
В теле метода stop()
```

Конечно же, в реальной программе методы, управляющие жизненным циклом приложения, никакой информации в выходной поток `System.out` обычно не выводят. Здесь это сделано лишь для того, чтобы было ясно, когда именно происходит вызов каждого метода. Кроме того, как ранее уже отмечалось, методы `init()` и `stop()` необходимо переопределять, только если при запуске и остановке приложения должны выполняться какие-то особые действия. В противном случае можно обойтись реализациями этих методов, предлагаемыми по умолчанию классом `Application`.

Перейдем к подробному рассмотрению программы. Она начинается с импорта четырех пакетов. Первым импортируется пакет `javafx.application`, в котором содержится класс `Application`. В пакете `javafx.scene` находится

класс Scene, а в пакете javafx.stage — класс Stage. Пакет javafx.scene.layout предоставляет ряд панелей компоновки. В программе используется панель FlowPane.

Далее создается класс приложения JavaFXSkel, расширяющий класс Application. Как уже отмечалось, Application — это класс, от которого наследуются все приложения JavaFX. Класс JavaFXSkel содержит четыре метода. Первый из них — метод main() — используется для загрузки приложения посредством вызова метода launch(). Обратите внимание на то, что методу launch() передается параметр args, принимаемый методом main(). Такой подход является обычным, однако методу launch() можно передать и другой набор параметров, в том числе пустой. Еще один важный момент: метод launch() требуется только автономным приложениям, во всех остальных случаях он не нужен. Однако в силу причин, указанных выше, все программы в этой главе включают как метод main(), так и метод launch().

Когда запускается приложение, исполняющая среда JavaFX в первую очередь вызывает метод init(). В данном случае этот метод просто выводит на консоль некоторое сообщение исключительно для того, чтобы сделать пример более наглядным. Обычно в нем выполняются все необходимые действия по инициализации приложения. Разумеется, если инициализация не требуется, то в переопределении метода init() нет никакой необходимости, поскольку по умолчанию всегда предоставляется его пустая реализация. Следует еще раз подчеркнуть, что метод init() не может быть использован для создания основной платформы или сцены GUI. Эти элементы должны конструироваться и отображаться в методе start().

Когда метод init() заканчивает свою работу, вызывается метод start(), в котором создается начальная сцена и устанавливается основное окно приложения. Проанализируем этот метод строка за строкой. Прежде всего обратите внимание на передаваемый ему параметр типа Stage. При вызове метода start() этот параметр получает ссылку на основную платформу приложения. Именно этот контейнер будет содержать сцену, используемую приложением.

После вывода на консоль сообщения, уведомляющего о начале работы метода start(), вызывается метод setTitle(), устанавливающий заголовок окна:

```
myStage.setTitle("Каркас приложения JavaFX");
```

Поступать так вовсе не обязательно, но в случае автономных приложений такая практика является общепринятой. Этот заголовок становится именем основного окна приложения.

На следующем этапе создается корневой узел сцены — единственный узел графа сцены, не имеющий родительского узла. В данном случае корневой узел — это объект типа FlowPane, но существуют и другие классы, которые могут служить таким узлом:

```
FlowPane rootNode = new FlowPane();
```

Как уже отмечалось, панель `FlowPane` использует плавающую компоновку. Этот тип компоновки характеризуется тем, что элементы последовательно располагаются в строках с автоматическим переходом на следующую строку, если для размещения очередного элемента в текущей строке не хватает места. (Следовательно, здесь мы имеем дело с тем же типом компоновки, что и в случае класса `FlowLayout`, входящего в библиотеки AWT и Swing.) В данном примере элементы компонуются построчно в горизонтальном направлении, однако возможна и компоновка по вертикальным столбцам. И хотя в данном приложении этого не требуется, существует возможность задания других свойств компоновки, таких как горизонтальный или вертикальный зазор между соседними элементами и их выравнивание.

В следующей строке кода корневой узел используется для создания объекта сцены:

```
Scene myScene = new Scene(rootNode, 300, 200);
```

Класс `Scene` имеет несколько конструкторов. Мы используем конструктор, который создает сцену с заданным корневым узлом и заданными значениями ширины и высоты.

```
Scene(Parent корень, double ширина, double высота)
```

Заметьте, что параметр *корень* имеет тип `Parent`. Этот класс является производным от класса `Node` и инкапсулирует узлы, у которых имеются дочерние узлы. Также обратите внимание на то, что для значений ширины и высоты задан тип `double`. В случае необходимости это позволяет передавать методу нецелочисленные значения. В данном примере корневым является узел `rootNode`, а значения ширины и высоты составляют соответственно 300 и 200.

В следующей строке программы объект `myScene` устанавливается в качестве сцены для платформы `myStage`:

```
myStage.setScene(myScene);
```

где `setScene()` — метод, определенный в классе `Stage`, который настраивает параметры сцены в соответствии с переданным ему аргументом.

В тех случаях, когда сцена в дальнейшем не используется, два предыдущих вызова могут быть объединены в один.

```
myStage.setScene(new Scene(rootNode, 300, 200));
```

В последующих примерах преимущество будет отдаваться именно этой форме вызова методов ввиду ее компактности.

Последняя инструкция метода `start()` отображает платформу и сцену.

```
myStage.show();
```

По сути, метод `show()` отображает окно, совместно создаваемое платформой и сценой.

При закрытии приложения его окно удаляется с экрана, и исполнительная среда JavaFX вызывает метод `stop()`. В данном случае этот метод выводит сообщение на консоль, тем самым подтверждая факт своего вызова. Однако в

реальных приложениях он, как правило, не выводит никакой информации. Кроме того, если не требуется выполнять какие-либо специальные действия при прекращении работы приложения, то отпадает и необходимость в переопределении метода `stop()`, поскольку его пустая реализация предоставляется по умолчанию.

Компиляция и выполнение программы JavaFX

Одним из преимуществ технологии JavaFX является то, что одна и та же программа способна выполняться в различных средах. Например, программа JavaFX может выполняться в виде автономного настольного приложения, в среде веб-браузера или в виде приложения Web Start. В то же время в некоторых случаях могут потребоваться различные вспомогательные файлы, например файл HTML или JNLP (Java Network Launch Protocol).

Вообще говоря, любая программа JavaFX компилируется подобно любой другой программе. Вместе с тем, в зависимости от целевой среды, может потребоваться выполнение ряда дополнительных действий. Поэтому зачастую самым простым способом является компиляция приложения JavaFX в какой-либо интегрированной среде разработки (Integrated Development Environment — IDE), обеспечивающей полную поддержку программирования в рамках технологии JavaFX. Если же вам нужно просто скомпилировать и протестировать JavaFX-приложения, представленные в данной главе, то это можно легко сделать средствами командной строки. Для этого достаточно скомпилировать и выполнить приложение, как это обычно делается с помощью команд `javac` и `java`. В результате вы получите автономное приложение, выполняющееся в настольной системе.

Поток выполнения приложения

В предыдущем обсуждении уже отмечалось, что метод `init()` не может быть использован для построения платформы или сцены. Эти элементы нельзя создавать и в конструкторе приложения. Причина заключается в том, что и платформа, и сцена должны формироваться в *потоке приложения*. В то же время конструктор приложения и метод `init()` вызываются в основном потоке, который также называют *стартовым потоком*. Вот почему их нельзя использовать для вызова конструкторов платформы и сцены. Вместо этого для создания начального графического интерфейса должен вызываться метод `start()`, как было сделано в примере, поскольку он вызывается в потоке приложения.

Более того, из потока приложения должны выполняться и любые изменения текущего состояния GUI. К счастью, в JavaFX события передаются программе через поток приложения. Поэтому для взаимодействия с графическим интерфейсом могут использоваться обработчики событий. Метод `stop()` также вызывается в потоке приложения.

Простой компонент JavaFX: Label

Основными составляющими большинства пользовательских интерфейсов являются компоненты, поскольку через них пользователь может взаимодействовать с приложением. Как и следовало ожидать, JavaFX предлагает богатый набор компонентов. Простейшим из них является метка, поскольку она просто отображает текстовое сообщение или графический элемент. Для наших целей метки удобны тем, что они просты в использовании и хорошо подходят для демонстрации методов построения графов сцены.

В JavaFX метка представляется экземпляром класса `Label`, входящего в пакет `javafx.scene.control`. Класс `Label` наследует свойства и методы от нескольких классов, включая `Labeled` и `Control`. Класс `Labeled` определяет ряд возможностей, общих для всех элементов с метками (т.е. тех, которые могут содержать текст), а класс `Control` — возможности, свойственные всем элементам управления.

Ниже мы будем использовать следующий конструктор класса `Label`:

```
Label(String строка);
```

Отображаемый текст задается параметром *строка*.

Созданную метку (или любой другой компонент) необходимо добавить к содержимому сцены, что означает ее добавление в граф сцены. Для этого следует прежде всего вызвать метод `getChildren()` для корневого узла графа сцены. Возвращаемое значение представляет собой список дочерних узлов типа `ObservableList<Node>`. Класс `ObservableList` находится в пакете `javafx.collections` и наследует класс `java.util.List`, являющийся частью библиотеки Java Collections Framework. Класс `List` определяет коллекцию, представляющую список объектов. Рассмотрение класса `List` и библиотеки Java Collections Framework выходит за рамки данной книги, просто подчеркнем, что класс `ObservableList` позволяет легко добавлять дочерние узлы. Это достигается вызовом метода `add()` для списка дочерних узлов, возвращенного методом `getChildren()`, с передачей ссылки на добавляемый узел, каковым в данном случае является метка.

Следующая программа представляет собой простое приложение JavaFX, отображающее метку.

```
// Демонстрация использования меток JavaFX

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class JavaFXLabelDemo extends Application {
```

```
    public static void main(String[] args) {

        // Запустить приложение JavaFX, вызвав метод launch()
        launch(args);
    }

    // Переопределить метод start()
    public void start(Stage myStage) {

        // Задать заголовок окна приложения
        myStage.setTitle("Использование метки JavaFX");

        // Использовать компоновку FlowPane для корневого узла
        FlowPane rootNode = new FlowPane();

        // Создать сцену
        Scene myScene = new Scene(rootNode, 300, 200);

        // Установить сцену на платформе
        myStage.setScene(myScene);
                                                     Создание метки
        // Создать метку
        Label myLabel = new Label("JavaFX - это мощный GUI");

        // Добавить метку в граф сцены                   Добавление метки
        rootNode.getChildren().add(myLabel); <---------  в граф сцены

        // Отобразить платформу вместе с ее сценой
        myStage.show();
    }
}
```

Спросим у эксперта

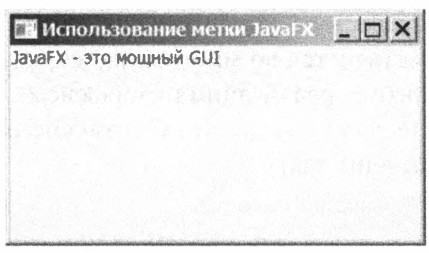
ВОПРОС. Вы рассказали о том, как добавить узел в граф сцены. Существуют ли способы удаления узлов из графа?

ОТВЕТ. Конечно. Чтобы удалить узел из графа сцены, следует вызвать метод `remove()` для объекта `ObservableList`. Например, вызов

```
rootNode.getChildren().remove(myLabel);
```

удаляет объект `myLabel` из сцены. Вообще говоря, класс `ObservableList` поддерживает широкий ряд методов управления списками. Вот лишь два примера. Чтобы определить, является ли список пустым, следует вызвать метод `isEmpty()`. Также можно определить количество узлов в списке, вызвав метод `size()`. Вам будет полезно самостоятельно исследовать возможности класса `ObservableList` в процессе изучения JavaFX.

Результат выполнения программы представлен на приведенном ниже рисунке.



В этой программе особого внимания заслуживает следующая строка:

```
rootNode.getChildren().add(myLabel);
```

Этот код добавляет метку в список узлов, по отношению к которым узел `rootNode` является родительским. По желанию эту строку можно было бы разделить на отдельные вызовы, но чаще всего она будет встречаться вам именно в таком виде.

Прежде чем продолжить, следует подчеркнуть, что класс `ObservableList` предоставляет метод `addAll()`, позволяющий добавить сразу два и более дочерних узла в граф сцены с помощью одного вызова. Пример будет показан далее.

Использование кнопок и событий

Программа, приведенная в предыдущем разделе, представляла собой простой пример использования компонента JavaFX и построения графа сцены, однако в ней не показано, как обрабатывать события. Обработка событий играет важную роль, поскольку большинство компонентов GUI генерирует события, которые обрабатываются пользовательскими программами. Например, когда вы используете кнопки, флажки или списки, все они генерируют события. Обработка событий в JavaFX во многом напоминает обработку событий в Swing, о которой шла речь в предыдущей главе, но выполняется гораздо проще. Одним из наиболее часто используемых компонентов является кнопка, поэтому ее события приходится обрабатывать чаще других. Следовательно, будет весьма полезно познакомиться с обработкой событий в JavaFX на примере кнопки.

Основные сведения о событиях

Базовым классом событий JavaFX является класс `Event`, находящийся в пакете `javafx.event`. Класс `Event` наследует класс `java.util.eventObject`, а это означает, что события JavaFX разделяют общую функциональность с другими событиями Java. Для класса `Event` определено несколько подклассов, из которых мы далее будем использовать только класс `ActionEvent`. Этот класс инкапсулирует события действий, генерируемые кнопкой.

Вообще говоря, подход к обработке событий JavaFX основан на модели делегатов. Чтобы обработать событие, сначала нужно зарегистрировать обработчик,

выступающий в качестве слушателя события. При наступлении какого-либо события вызывается соответствующий слушатель, который должен отреагировать на событие и после этого вернуть управление. В этом отношении управление событиями JavaFX осуществляется во многом так же, как и событиями Swing.

Обработка событий требует реализации интерфейса `EventHandler`, который также находится в пакете `javafx.event`. Синтаксис объявления этого обобщенного интерфейса выглядит так:

```
Interface EventHandler<T extends Event>
```

где `T` задает тип обрабатываемого события. Данный интерфейс определяет один метод, `handle()`, которому передается объект события в качестве параметра.

```
void handle(T объект_события)
```

В данном случае *объект_события* — это сгенерированное событие. Обычно обработчики событий реализуются посредством использования анонимных внутренних классов или лямбда-выражений, но для этого могут использоваться и независимые классы, если такое решение больше подходит для конкретного приложения (например, в тех случаях, когда один и тот же обработчик должен обслуживать события, поступающие от разных источников).

Компонент Button

В JavaFX кнопка представлена классом `Button`, который находится в пакете `javafx.scene.control`. Список классов, которые наследует класс `Button`, довольно внушителен и включает такие классы, как `ButtonBase`, `Labeled`, `Region`, `Control`, `Parent` и `Node`. Если вы обратитесь к разделам документации API, относящимся к классу `Button`, то убедитесь в том, что бо́льшая часть его функциональности унаследована от базовых классов. Кроме того, он предлагает широкий ряд возможностей выбора. Однако мы будем использовать его форму, предоставляемую по умолчанию. Кнопки могут содержать текст, графику или и то и другое одновременно. В нашем примере мы будем использовать текстовые кнопки.

Синтаксис используемого нами конструктора класса `Button` таков:

```
Button (String строка)
```

В данном случае *строка* — это текст, отображаемый на кнопке.

После щелчка на кнопке генерируется событие `ActionEvent`. Класс `ActionEvent` находится в пакете `javafx.event`. Регистрация слушателя этого события осуществляется посредством вызова метода `setOnAction()` для кнопки. Общая форма объявления этого метода выглядит так:

```
final void setOnAction(EventHandler<ActionEvent> обработчик)
```

где *обработчик* — это обработчик, подлежащий регистрации. Как уже упоминалось, вы будете часто использовать анонимные внутренние классы для обработчиков. Метод `setOnAction()` устанавливает свойство `OnAction`, в котором хранится ссылка на обработчик. Как и при обработке любого другого события

Java, обработчик событий кнопки должен как можно быстрее отреагировать на событие и после этого немедленно вернуть управление. Если выполнение обработчика занимает слишком много времени, это будет серьезно замедлять работу приложения. Для выполнения длительных операций следует использовать отдельные потоки выполнения.

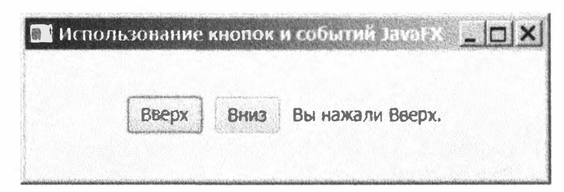
Обработка событий кнопки

В приведенной ниже программе демонстрируется обработка событий компонента `Button`. Программа отображает две кнопки, **Вверх** и **Вниз**, и метку. Всякий раз, когда нажимается кнопка, метка отображает текст, указывающий на то, какая из кнопок была нажата. Таким образом, по своей функциональности эта программа аналогична программе, демонстрирующей использование компонента `JButton`, которую мы рассмотрели в предыдущей главе. Возможно, вам будет интересно провести самостоятельный сравнительный анализ этих программ.

```
// Демонстрация обработки событий JavaFX для кнопок

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

   Label response;

   public static void main(String[] args) {

      // Запустить приложение JavaFX, вызвав метод launch()
      launch(args);
   }

   // Переопределить метод start()
   public void start(Stage myStage) {

      // Задать заголовок окна приложения
      myStage.setTitle("Использование кнопок и событий JavaFX");

      // Использовать компоновку FlowPane для корневого узла.
      // В данном случае величина вертикального и горизонтального
      // зазоров составляет 10.
      FlowPane rootNode = new FlowPane(10, 10);

      // Центрировать компоненты на сцене
      rootNode.setAlignment(Pos.CENTER);

      // Создать сцену
      Scene myScene = new Scene(rootNode, 300, 100);
```

```
        // Установить сцену на платформе
        myStage.setScene(myScene);

        // Создать метку
        response = new Label("Нажмите кнопку");

        // Создать две кнопки
        Button btnUp = new Button("Вверх");  ◄──┐
        Button btnDown = new Button("Вниз"); ◄──┴── Создание двух кнопок

        // Обработать события действий для кнопки "Вверх"
        btnUp.setOnAction(new EventHandler<ActionEvent>() {  ◄──┐
           public void handle(ActionEvent ae) {                 │
              response.setText("Вы нажали Вверх.");             │
           }                                                    │
        });                                                     ├── Создание обработчиков
                                                                │   событий для кнопок
        // Обработать события действий для кнопки "Вниз"        │
        btnDown.setOnAction(new EventHandler<ActionEvent>() { ◄─┘
           public void handle(ActionEvent ae) {
              response.setText("Вы нажали Вниз.");
           }
        });

        // Добавить метку и кнопки в граф сцены
        rootNode.getChildren().addAll(btnUp, btnDown, response);

        // Отобразить платформу вместе с ее сценой
        myStage.show();
    }
}
```

Результат выполнения программы представлен на приведенном ниже рисунке.



Проанализируем ключевые места данной программы. Прежде всего обратите внимание на следующие строки.

```
Button btnUp = new Button("Вверх");
Button btnDown = new Button("Вниз");
```

Этот код создает две текстовые кнопки. На первой из них отображается текст `Вверх`, на второй — `Вниз`.

После этого для каждой из кнопок устанавливается обработчик событий действий. Вот как выглядит соответствующий код для кнопки `Вверх`.

```
// Обработать события действий для кнопки "Вверх"
btnUp.setOnAction(new EventHandler<ActionEvent>() {
```

```
    public void handle(ActionEvent ae) {
        response.setText("Вы нажали Вверх.");
    }
});
```

Как уже было сказано, кнопки реагируют на события типа ActionEvent. Чтобы зарегистрировать обработчик этих событий, следует вызвать метод setOnAction() для соответствующей кнопки. Интерфейс EventHandler реализуется с использованием анонимного внутреннего класса. (Вспомните, что в классе EventHandler определен только метод handle().) В теле метода handle() задается текст, который будет отображаться меткой response и тем самым иллюстрировать нажатие кнопки **Вверх**. Это делается посредством вызова метода setText() для метки. Точно так же обрабатываются события кнопки **Вниз**.

После установки обработчиков событий метка и обе кнопки добавляются в граф сцены с помощью метода addAll().

```
rootNode.getChildren().addAll(btnUp, btnDown, response);
```

Метод addAll() добавляет переданный ему список узлов в вызывающий родительский узел. Разумеется, для добавления этих узлов можно было бы использовать три отдельных вызова метода add(), но в данной ситуации удобнее использовать метод addAll().

Можно отметить еще два интересных момента, имеющих отношение к отображению компонентов в окне. Во-первых, корневой узел создается следующей инструкцией:

```
FlowPane rootNode = new FlowPane(10, 10);
```

где конструктору объекта типа FlowPane передаются два значения, устанавливающих величину горизонтального и вертикального зазоров, которые будут оставлены вокруг элементов при их размещении на сцене. Если не указать эти значения, то два соседних элемента (например, две кнопки) расположатся на сцене вплотную друг к другу. В таком случае элементы сольются на экране, и их будет трудно различать, что сделает пользовательский интерфейс весьма неудобным. Задание зазоров позволяет избавиться от этого недостатка.

Во-вторых, заслуживает внимания следующая строка кода, устанавливающая способ выравнивания компонентов при их компоновке на панели FlowPane:

```
rootNode.setAlignment(Pos.CENTER);
```

Здесь элементы выравниваются по центру за счет вызова метода set Alignment() для панели FlowPane. Значение Pos.CENTER указывает на то, что центрирование осуществляется как по вертикали, так и по горизонтали. Возможны и другие способы выравнивания. Pos — это перечисление, содержащее список констант выравнивания. Оно находится в пакете javafx.geometry.

Прежде чем продолжить, нужно сделать еще одно замечание. В предыдущей программе для обработки событий кнопки использовался анонимный внутренний класс. Но в связи с тем что интерфейс EventHandler определяет только один абстрактный метод, handle(), вместо этого можно передать методу

setOnAction() лямбда-выражение. В качестве примера ниже приведен модифицированный вариант обработчика событий для кнопки **Вверх**, в котором используется лямбда-выражение.

```
btnUp.setOnAction( (ae) ->
                            response.setText("Вы нажали Вверх.")
                  );
```

Как видите, лямбда-выражение более компактно по сравнению с анонимным внутренним классом. (Вы будете использовать лямбда-выражение, видоизменяя эту программу в процессе выполнения упражнения 10, приведенного в конце главы.)

Три других компонента JavaFX

В JavaFX определен богатый набор компонентов, которые содержатся в пакете javafx.scene.control. С двумя из них вы уже знакомы: это компоненты Label и Button. На очереди следующие три: CheckBox, ListView и TextField. Как несложно понять, они реализуют флажок, список и текстовое поле соответственно. С их помощью мы продемонстрируем ряд популярных методик работы с компонентами. Когда вы разберетесь с простыми элементами управления, вы сможете самостоятельно изучить все остальные компоненты.

Функциональность описанных ниже компонентов аналогична функциональности компонентов Swing, которым была посвящена предыдущая глава. В процессе изучения данного раздела вам будет полезно сравнить способы реализации компонентов в библиотеках JavaFX и Swing.

Компонент CheckBox

В JavaFX функциональность флажка инкапсулирует класс CheckBox, являющийся непосредственным потомком класса ButtonBase. Таким образом, он является особым типом кнопки. Учитывая широкое использование флажков, можно не сомневаться, что они вам уже знакомы, однако флажки JavaFX немного сложнее, чем кажутся на первый взгляд. Это обусловлено тем, что класс CheckBox поддерживает три состояния. Два из них очевидны — это состояния "установлен" и "снят", которые соответствуют поведению по умолчанию. Третье состояние называется недетерминированным (или неопределенным). Обычно оно используется для индикации того, что состояние флажка не было установлено или оно не имеет значения в данной конкретной ситуации. Чтобы это состояние можно было использовать, его необходимо активизировать явным образом. Соответствующая процедура будет продемонстрирована в упражнении 17.1. А пока мы сосредоточимся на традиционном поведении флажка.

Далее мы будем использовать конструктор класса CheckBox следующего вида:

```
CheckBox(String строка)
```

Этот конструктор создает флажок с пояснительной надписью, текст которой задается параметром *строка*. Как и в случае других разновидностей кнопок, при выборе флажка CheckBox генерируется событие действия.

Использование флажков продемонстрировано в приведенной ниже программе. Она отображает четыре флажка, представляющих различные типы компьютеров, которые обозначены как Смартфон, Планшет, Ноутбук и ПК. Всякий раз, когда изменяется состояние флажка, генерируется событие действия. Обработка события заключается в отображении состояния флажка (установлен или снят), а также списка установленных флажков.

```
// Демонстрация использования флажков

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class CheckboxDemo extends Application {

   CheckBox cbSmartphone;
   CheckBox cbTablet;
   CheckBox cbNotebook;
   CheckBox cbDesktop;

   Label response;
   Label selected;

   String computers;

   public static void main(String[] args) {

     // Запустить приложение JavaFX, вызвав метод launch()
     launch(args);
   }

   // Переопределить метод start()
   public void start(Stage myStage) {

      // Задать заголовок окна приложения
      myStage.setTitle("Демонстрация флажков");

      // Использовать компоновку FlowPane для корневого узла.
      // В данном случае величина вертикального и горизонтального
      // зазоров составляет 10.
      FlowPane rootNode = new FlowPane(Orientation.VERTICAL, 10, 10);

      // Центрировать компоненты на сцене
      rootNode.setAlignment(Pos.CENTER);
```

```
// Создать сцену
Scene myScene = new Scene(rootNode, 230, 200);

// Установить сцену на платформе
myStage.setScene(myScene);

Label heading = new Label("Какие у вас есть устройства?");

// Создать метку, извещающую об изменении состояния флажка
response = new Label("");

// Создать метку, извещающую о выборе любого флажка
selected = new Label("");

// Создать флажки
cbSmartphone = new CheckBox("Смартфон");
cbTablet = new CheckBox("Планшет");
cbNotebook = new CheckBox("Ноутбук");
cbDesktop = new CheckBox("ПК");

// Обработка событий действий для флажков
cbSmartphone.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
      if(cbSmartphone.isSelected())
         response.setText("Был выбран смартфон.");
      else
         response.setText("Выбор смартфона отменен.");

      showAll();
   }
});

cbTablet.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
      if(cbTablet.isSelected())
         response.setText("Был выбран планшет.");
      else
         response.setText("Выбор планшета отменен.");

      showAll();
   }
});

cbNotebook.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
      if(cbNotebook.isSelected())
         response.setText("Был выбран ноутбук.");
      else
         response.setText("Выбор ноутбука отменен.");

      showAll();
   }
});

cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
```

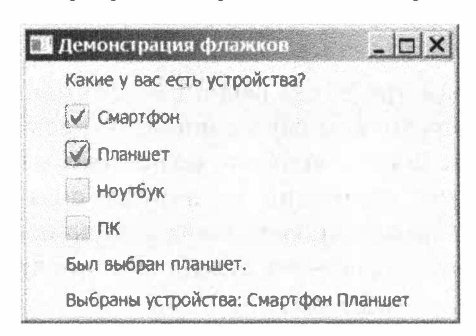
Создание флажков

Обработка событий флажков

```
         if(cbDesktop.isSelected())
            response.setText("Был выбран ПК.");
         else
            response.setText("Выбор ПК отменен.");

         showAll();
      }
   });

   // Добавить компоненты в граф сцены
   rootNode.getChildren().addAll(heading, cbSmartphone, cbTablet,
                                 cbNotebook, cbDesktop, response,
                                 selected);

   // Отобразить платформу вместе с ее сценой
   myStage.show();

   showAll();
 }

 // Обновить и отобразить варианты выбора
 void showAll() {
   computers = "";
   if(cbSmartphone.isSelected()) computers = "Смартфон ";
   if(cbTablet.isSelected()) computers += "Планшет ";
   if(cbNotebook.isSelected()) computers += "Ноутбук ";
   if(cbDesktop.isSelected()) computers += "ПК";

   selected.setText("Выбраны устройства: " + computers);
 }
}
```

Использование метода `isSelected()` для определения состояния флажков

Результат выполнения программы представлен на приведенном ниже рисунке.



В работе этой программы нет ничего сложного. Всякий раз, когда изменяется состояние флажка, генерируется событие `ActionEvent`. Сначала обработчики событий сообщают о том, установлен флажок или снят. С этой целью для источника события вызывается метод `isSelected()`. Возвращаемому значению `true` соответствует установка флажка, значению `false` — снятие. После этого вызывается метод `showAll()`, который выводит список всех установленных флажков.

В этой программе можно отметить еще один интересный момент. Обратите внимание на то, что в ней используется вертикальная компоновка.

```
FlowPane rootNode = new FlowPane(Orientation.VERTICAL, 10, 10);
```

По умолчанию компоненты размещаются на панели FlowPane по горизонтали. Для создания панели с вертикальной компоновкой следует передать конструктору значение Orientation.VERTICAL в качестве первого аргумента.

Упражнение 17.1 Использование неопределенного состояния компонента CheckBox

CheckBoxDemo.java

Как уже отмечалось, по умолчанию компонент CheckBox реализует два состояния, соответствующие установленному и снятому флажку. Но он поддерживает и третье, неопределенное, состояние, которое можно использовать для индикации того, что состояние флажка еще не устанавливалось или же что эта возможность в данной ситуации неприменима. Неопределенное состояние флажка необходимо активизировать явным образом, так как оно не предоставляется по умолчанию. Кроме того, обработчик событий флажка должен обрабатывать и неопределенное состояние. Выполним упражнение, которое проиллюстрирует этот процесс. Суть упражнения состоит в том, что в предыдущую программу CheckBoxDemo, в компонент CheckBox, соответствующий смартфону, добавляется поддержка неопределенного состояния. Поэтапное описание процесса создания модифицированного варианта программы приведено ниже.

1. Чтобы активизировать неопределенное состояние флажка, необходимо вызвать метод setAllowIndeterminate().

   final void setAllowIndeterminate(boolean *режим*)

   Если значение параметра *режим* равно true, активизируется неопределенное состояние. В противном случае оно деактивизируется. Когда неопределенное состояние активизировано, пользователь может переводить компонент в одно из трех состояний: установлен, снят и не определено. Следовательно, чтобы активизировать неопределенное состояние для флажка **Смартфон**, необходимо добавить следующую строку кода:

   ```
   cbSmartphone.setAllowIndeterminate(true);
   ```

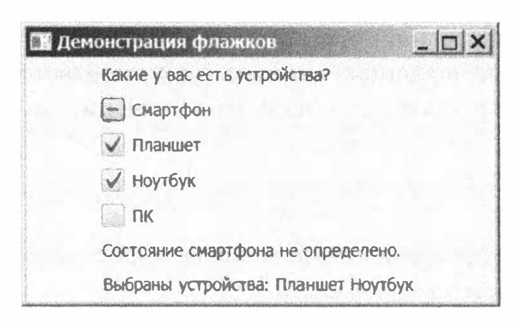
2. Чтобы определить, находится ли флажок в неопределенном состоянии, нужно вызвать метод isIndeterminate().

   ```
   final boolean isIndeterminate()
   ```

   Этот метод возвращает значение true, если флажок находится в неопределенном состоянии, и значение false в противном случае. Обработчик событий флажка теперь должен тестировать и неопределенное состояние.

```
cbSmartphone.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
      if(cbSmartphone.isIndeterminate())
         response.setText("Состояние смартфона не определено.");
      else if(cbSmartphone.isSelected())
         response.setText("Был выбран смартфон.");
      else
         response.setText("Выбор смартфона отменен.");

      showAll();
   }
});
```

3. Внеся указанные изменения, скомпилируйте и выполните программу. Теперь, как показано на приведенном ниже рисунке, вы сможете устанавливать флажок **Смартфон** в неопределенное состояние.



Компонент ListView

Другой распространенный компонент — список, функциональность которого в JavaFX инкапсулирует класс `ListView`, отображающий список элементов с возможностью выбора одного или нескольких из них. Полезным свойством списка `ListView` является автоматическое добавление полос прокрутки, если количество элементов списка таково, что не все они могут быть одновременно отображены в поле списка. Эта способность списка `ListView` обеспечивать эффективное использование дефицитного экранного пространства сделала его весьма популярным на фоне остальных компонентов, предоставляющих возможности выбора.

Класс `ListView` — обобщенный и имеет следующую форму объявления:

```
class ListView<T>
```

где `T` обозначает тип элементов, хранящихся в списке. Чаще всего это элементы типа `String`, но допускаются и другие.

Далее мы будем использовать следующий вариант конструктора `ListView`:

```
ListView(ObservableList<T> список)
```

Список элементов, подлежащих отображению, задается параметром *список*, который представляет собой объект типа `ObservableList`. Как уже отмечалось,

класс `ObservableList` поддерживает список объектов. По умолчанию класс `ListView` позволяет выбирать только один элемент списка в каждый момент времени. Также предусмотрен режим группового выбора, но мы будем использовать установленный по умолчанию режим выбора одиночных элементов.

Вероятно, простейший способ создания объекта `ObservableList` для списка `ListView` — использование метода `observableArrayList()`, работающего по принципу фабрики объектов. Он определен как статический метод в классе `FXCollections` (находится в пакете `javafx.collections`). Мы будем использовать следующую версию этого метода:

```
static <E> ObservableList<E> observableArrayList(E ... элементы)
```

В данном случае `E` обозначает тип элементов, которые передаются посредством параметра *элементы*.

Компонент `ListView` предлагает заданные по умолчанию значения ширины и высоты списка, однако в некоторых случаях желательно установить другие предпочтительные значения, которые лучше соответствуют вашим потребностям. Одним из способов установки этих значений является вызов методов `setPrefHeight()` и `setPrefWidth()`.

```
final void setPrefHeight(double высота)
final void setPrefWidth(double высота)
```

Возможно и другое решение, позволяющее задать одновременно оба размера с помощью вызова метода `setPrefSize()`.

```
void setPrefSize(double ширина, double высота)
```

Список `ListView` можно использовать двояким образом. Во-первых, можно игнорировать события, генерируемые списком, ограничиваясь получением выбранного элемента списка, когда это понадобится программе. Во-вторых, можно вести мониторинг изменений, происходящих в списке, зарегистрировав обработчик событий. Это позволит реагировать на каждую смену выбора элемента списка. Именно такой подход применяется далее.

Слушатель событий смены выбранного элемента списка поддерживается интерфейсом `ChangeListener`, который находится в пакете `javafx.bean.value`. Интерфейс `ChangeListener` определяет только один метод, `changed()`:

```
void changed(ObservableValue<? extends T> changed, T oldVal, T newVal)
```

где *changed* — экземпляр класса `ObservableValue<T>`, инкапсулирующий объект, за изменениями состояния которого можно наблюдать. Через параметры *oldVal* и *newVal* методу передаются прежнее и новое значения соответственно. Таким образом, в данном случае в *newVal* хранится ссылка на только что выбранный элемент списка.

Чтобы прослушивать события, прежде всего необходимо получить модель выбора, используемую компонентом `ListView`. Это можно сделать, вызвав для списка метод `getSelectionModel()`:

```
final MultipleSelectionModel<T> getSelectionModel()
```

Данный метод возвращает ссылку на модель. Класс MultipleSelectionModel определяет модель, используемую при групповом выборе, и наследует класс SelectionModel. Однако групповой выбор разрешен в списке ListView только в том случае, если режим группового выбора активизирован.

Используя модель, возвращенную методом getSelectionModel(), вы сможете получить ссылку на свойство выбранного элемента, которое определяет, что именно должно происходить, когда выбирается элемент списка. Для этого следует вызвать метод selectedItemProperty():

```
final ReadOnlyObjectProperty<T> selectedItemProperty()
```

Добавление слушателя событий изменений в это свойство осуществляется путем вызова метода addListener() для возвращенного свойства. Метод addListener() имеет следующий синтаксис:

```
void addListener(ChangeListener<? super T> слушатель)
```

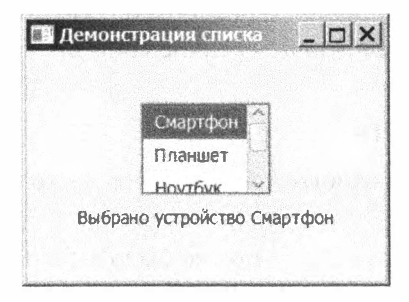
В данном случае T обозначает тип свойства.

Следующий пример реализует на практике все вышесказанное. В нем создается список типов компьютеров, обеспечивающий возможность выбора нужного элемента. При выборе какого-либо элемента отображается соответствующий пояснительный текст.

```
// Демонстрация использования списка

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.beans.value.*;
import javafx.collections.*;

public class ListViewDemo extends Application {

   Label response;

   public static void main(String[] args) {

      // Запустить приложение JavaFX, вызвав метод launch()
      launch(args);
   }

   // Переопределить метод start()
   public void start(Stage myStage) {

      // Задать заголовок окна приложения
      myStage.setTitle("Демонстрация списка");

      // Использовать компоновку FlowPane для корневого узла.
      // В данном случае величина вертикального и горизонтального
```

```
        // зазоров составляет 10.
        FlowPane rootNode = new FlowPane(10, 10);

        // Центрировать компоненты на сцене
        rootNode.setAlignment(Pos.CENTER);

        // Создать сцену
        Scene myScene = new Scene(rootNode, 200, 120);

        // Установить сцену на платформе
        myStage.setScene(myScene);

        // Создать метку
        response = new Label("Выбор типа устройства");

        // Создать объект типа ObservableList для списка
        ObservableList<String> computerTypes =
           FXCollections.observableArrayList("Смартфон", "Планшет",
                                             "Ноутбук", "ПК" );

        // Создать список
        ListView<String> lvComputers =
                        new ListView<String>(computerTypes);
                            ▲
                            └─ Создание списка, который отображает элементы
                               из объекта computerTypes
        // Задать предпочтительные значения высоты и ширины
        lvComputers.setPrefSize(100, 70);

        // Получить модель выбора для списка
        MultipleSelectionModel<String> lvSelModel =
                          lvComputers.getSelectionModel();

        // Использовать слушатель для реагирования на изменения
        // выделения внутри списка
        lvSelModel.selectedItemProperty().addListener(          Обработка событий
                          new ChangeListener<String>() { ◄── изменения
           public void changed(ObservableValue<? extends String>
                              changed, String oldVal, String newVal) {

              // Отобразить выбор
              response.setText("Выбрано устройство " + newVal);
           }
        });

        // Добавить метку и список в граф сцены
        rootNode.getChildren().addAll(lvComputers, response);

        // Отобразить платформу вместе с ее сценой
        myStage.show();
    }
}
```

Результат выполнения программы представлен на приведенном ниже рисунке.



Обратите внимание на вертикальную полосу прокрутки, позволяющую просмотреть все элементы списка. Как уже отмечалось, полоса прокрутки автоматически добавляется в тех случаях, когда не все элементы списка могут отображаться одновременно. Это делает список `ListView` чрезвычайно удобным.

Особого внимания заслуживает способ конструирования списка `ListView` в программе. Сначала создается объект `ObservableList`.

```
ObservableList<String> computerTypes =
   FXCollections.observableArrayList("Смартфон", "Планшет",
                                     "Ноутбук", "ПК" );
```

В данном случае для создания списка строк применяется метод `observableArrayList()`. После этого объект `ObservableList` используется для инициализации списка `ListView`:

```
ListView<String> lvComputers = new ListView<String>(computerTypes);
```

Затем в программе устанавливаются предпочтительные значения ширины и высоты компонента.

Обратите внимание на способ получения модели выбора для списка `lvComputers`.

```
MultipleSelectionModel<String> lvSelModel =
   lvComputers.getSelectionModel();
```

Как уже отмечалось, объект `ListView` использует модель `MultipleSelectionModel`, даже если разрешен только режим выбора одиночных элементов. Далее для модели вызывается метод `SelectedItemProperty()` и регистрируется слушатель событий.

```
lvSelModel.selectedItemProperty().addListener(
                                   new ChangeListener<String>() {
   public void changed(ObservableValue<? extends String> changed,
                       String oldVal, String newVal) {

      // Отобразить выбор
      response.setText("Выбрано устройство " + newVal);
   }
});
```

Вам будет интересно узнать, что тот же самый базовый механизм, который используется для прослушивания и обработки событий изменений, применим к любому компоненту, генерирующему такие события.

Спросим у эксперта

ВОПРОС. Как разрешить одновременный выбор нескольких элементов в списке `ListView`?

ОТВЕТ. Чтобы в списке `ListView` можно было выбирать сразу несколько элементов, необходимо установить режим группового выбора. Это делается посредством вызова метода `setSelectionMode()` для модели `ListView`. Общая форма объявления данного метода представлена ниже.

```
final void setSelectionMode(SelectionMode режим)
```

Здесь параметр *режим* может иметь одно из двух значений: `SelectionMode.MULTIPLE` и `SelectionMode.SINGLE`. Режиму группового выбора соответствует значение `SelectionMode.MULTIPLE`.

Одним из способов получения списка выбранных элементов является вызов метода `getSelectedItems` для модели выбора. Вот общая форма объявления этого метода:

```
ObservableList<T> getSelectedItems()
```

Метод возвращает выбранные элементы в виде списка типа `ObservableList`. Для поэлементного просмотра возвращенного списка можно воспользоваться циклом типа `for-each`.

Компонент `TextField`

Несмотря на очевидную полезность компонентов `Button`, `CheckBox` и `ListView`, все они реализуют средства, обеспечивающие возможность выбора лишь заранее определенных параметров или действий. Однако иногда желательно, чтобы пользователь имел возможность самостоятельно вводить строку по собственному усмотрению. В JavaFX предусмотрено несколько текстовых компонентов, обеспечивающих этот тип ввода. К их числу относится компонент `TextField`, предназначенный для ввода однострочного текста. Такая возможность оказывается удобной при вводе имен, идентификаторов, адресов и т.п. Как и все текстовые компоненты, класс `TextField` наследует класс `TextInputControl`, который определяет значительную часть его функциональности.

В классе `TextField` определены два конструктора. Один из них является конструктором по умолчанию и создает пустое текстовое поле стандартного размера. Второй позволяет задать начальное содержимое текстового поля. Далее мы будем использовать конструктор, заданный по умолчанию.

В некоторых случаях можно ограничиться использованием стандартного размера текстового поля, заданного по умолчанию, однако часто возникает необходимость устанавливать размер поля по собственному усмотрению. Это делается путем вызова метода `setPrefColumnCount()`, имеющего следующий синтаксис:

```
final void setPrefColumnCount(int столбцы)
```

Параметр *столбцы* позволяет устанавливать размер компонента `TextField`.

Содержимое текстового поля можно задать, вызвав метод `setText()`. Для получения текущего содержимого можно вызвать метод `getText()`. Кроме этих двух основных операций, компонент `TextField` поддерживает ряд других возможностей, таких как вырезание, вставка и присоединение текста, которые вам будет полезно изучить самостоятельно. Также предоставляется возможность выделения части текста программными средствами.

Одна из наиболее интересных особенностей компонента `TextField` состоит в том, что он позволяет предоставлять текст-подсказку, который будет отображаться в текстовом поле до тех пор, пока пользователь не начнет вводить какое-либо значение. Это делается с помощью метода `setPromptText()`, синтаксис которого таков:

```
final void setPromptText(String строка)
```

где параметр *строка* — это строка, отображаемая в текстовом поле, когда никакой другой текст еще не введен. На экране эта строка отображается с пониженной яркостью (затенена).

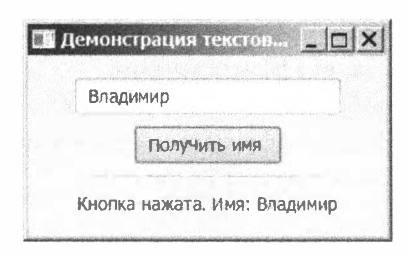
Если пользователь нажимает клавишу <Enter> в то время, когда фокус ввода находится в поле компонента `TextField`, генерируется событие действия. В одних случаях обработка таких событий может быть полезной, тогда как в других программах требуется всего лишь получить введенный текст, не обрабатывая события. В следующей программе демонстрируются оба подхода. В программе создается текстовое поле, запрашивающее ввод имени. Если пользователь нажмет клавишу <Enter> в то время, когда фокус ввода находится в поле ввода, или щелкнет на кнопке **Ввести имя**, строка будет передана программе, которая выведет соответствующий текст. Обратите внимание на использование в программе текстовой подсказки.

```
// Демонстрация использования текстового поля

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class TextFieldDemo extends Application {
```

```
TextField tf;
Label response;

public static void main(String[] args) {

   // Запустить приложение JavaFX, вызвав метод launch()
   launch(args);
}

// Переопределить метод start()
public void start(Stage myStage) {

   // Задать заголовок окна приложения
   myStage.setTitle("Демонстрация текстового поля");

   // Использовать компоновку FlowPane для корневого узла.
   // В данном случае величина вертикального и горизонтального
   // зазоров составляет 10.
   FlowPane rootNode = new FlowPane(10, 10);

   // Центрировать компоненты на сцене
   rootNode.setAlignment(Pos.CENTER);

   // Создать сцену
   Scene myScene = new Scene(rootNode, 230, 140);

   // Установить сцену на платформе
   myStage.setScene(myScene);

   // Создать метку
   response = new Label("Получить имя: ");

   // Создать кнопку, управляющую получением текста
   Button btnGetText = new Button("Получить имя");

   // Создать текстовое поле
   tf = new TextField(); ◄——— Создание текстового поля

   // Задать подсказку
   tf.setPromptText("Введите имя."); ◄——— Настройка подсказки
                                          для текстового поля

   // Задать предпочтительное количество столбцов
   tf.setPrefColumnCount(15); ◄——— Настройка ширины
                                   текстового поля

   // Использовать лямбда-выражение, обрабатывающее
   // события действий для текстового поля. События
   // действий генерируются при нажатии клавиши <ENTER>
   // в то время, когда фокус ввода находится в текстовом
   // поле. В данном случае обработка события заключается
   // в получении и отображении текста.
   tf.setOnAction( (ae) -> response.setText("Введено. Имя: " +
                                      tf.getText()));
                              ▲
              Установка событий действий для текстового поля
```

```
        // Использовать лямбда-выражение для получения текста из
        // текстового поля при нажатии кнопки
        btnGetText.setOnAction((ae) ->
           response.setText("Кнопка нажата. Имя: " + tf.getText()));

        // Использовать разделитель для лучшей организации вывода
        Separator separator = new Separator();
        separator.setPrefWidth(180);

        // Добавить компоненты в граф сцены
        rootNode.getChildren().addAll(tf, btnGetText, separator,
                                      response);

        // Отобразить платформу вместе с ее сценой
        myStage.show();
    }
}
```

Результат выполнения программы представлен на приведенном ниже рисунке.



Обратите внимание на использование лямбда-выражений в обработчиках событий. Каждый из обработчиков включает единственный вызов метода, поэтому все они являются идеальными кандидатами для использования лямбда-выражений.

Спросим у эксперта

ВОПРОС. Какие еще текстовые компоненты поддерживаются в JavaFX?

ОТВЕТ. К числу других текстовых компонентов относится компонент `TextArea`, который поддерживает ввод многострочного текста, и `PasswordField`, предназначенный для ввода паролей. Возможно, вас заинтересуют также компоненты `HTMLEditor` и `TextInputDialog`.

Знакомство с эффектами и преобразованиями

Главное преимущество библиотеки JavaFX — предоставление возможности детально контролировать внешний вид компонентов (и вообще любых узлов графа сцены) за счет применения к ним *эффектов* и *преобразований*. Как эффекты, так и преобразования способны придать графическому интерфейсу вашего приложения изысканный современный вид, на который рассчитывают

пользователи. Как вы увидите далее, легкость использования этих средств библиотеки JavaFX является одной из самых сильных ее сторон. Тема эффектов и преобразований слишком обширна, чтобы мы могли рассмотреть ее более подробно, однако даже краткого введения в нее вам будет достаточно для того, чтобы понять, какие преимущества предоставляются этими средствами.

Эффекты

Эффекты поддерживаются абстрактным классом `Effect` и его конкретными подклассами, которые находятся в пакете `javafx.scene.effect`. С помощью эффектов вы сможете придать узлу графа сцены внешний вид, в точности соответствующий вашим предпочтениям. Библиотека предоставляет ряд встроенных эффектов, часть которых перечислена в следующей таблице.

`Bloom`	Увеличивает яркость более светлых частей узла
`BoxBlur`	Размывает изображение узла
`DropShadow`	Отображает тени, отбрасываемые узлами
`Glow`	Создает эффект свечения
`InnerShadow`	Отображает тень внутри узла
`Lighting`	Создает эффекты теней при наличии источника света
`Reflection`	Создает эффекты отражения

Эти и другие эффекты просты в использовании и применимы к любому узлу, включая компоненты. Разумеется, в зависимости от особенностей компонентов, некоторые эффекты могут оказываться более полезными, чем другие.

Чтобы установить эффект для узла, следует вызвать метод `setEffect()`, определяемый классом `Node`.

```
final void setEffect(Effect эффект)
```

где параметр *эффект* — это применяемый эффект. Для указания отсутствия эффекта передается значение `null`. Таким образом, для добавления эффекта в узел необходимо сначала создать экземпляр эффекта, а затем передать его методу `setEffect()`. После этого эффект будет использоваться каждый раз при визуализации узла (при условии, что данный эффект поддерживается средой выполнения). Возможности этих средств будут продемонстрированы на примере двух эффектов: `Reflection` и `BoxBlur`. Однако сама процедура в основном не зависит от того, какой эффект выбран.

Эффект `BoxBlur` размывает изображение узла, к которому он применен. Он называется так потому, что используемая при этом техника размытия основана на корректировке пикселей, ограниченных прямоугольной областью. Степень размытия можно контролировать. Чтобы использовать эффект размытия, необходимо создать экземпляр `BoxBlur`. Класс `BoxBlur` определяет два конструктора. Мы будем использовать конструктор следующего вида:

```
BoxBlur(double ширина, double высота, int итерации)
```

где параметры *ширина* и *высота* определяют размеры прямоугольной области, в пределах которой будут размываться пиксели. Каждый из этих параметров может иметь значения в диапазоне 0–255, включая граничные значения. Как правило, значения выбираются в нижней части указанного диапазона. Параметр *итерации* определяет кратность применения эффекта, и его величина должна выбираться в пределах от 0 до 3, включая граничные значения. Также поддерживается конструктор по умолчанию, которому соответствуют следующие значения параметров: ширина и высота — 5.0, итерации — 1.

Ширину и высоту прямоугольной области для уже созданного экземпляра BoxBlur можно изменить с помощью методов setWidth() и setHeight().

```
final void setWidth(double ширина)
final void Height(double высота)
```

Количество итераций можно изменить с помощью метода setIterations():

```
final void setIterations(double итерации)
```

Использование этих трех методов позволяет изменить параметры эффекта размытия во время выполнения программы.

Эффект Reflection позволяет получить отражение узла, для которого он вызывается, по вертикали. Особенно часто этот эффект применяется к текстовым объектам, таким как метки. Эффект Reflection позволяет контролировать внешний вид отраженного узла. Например, он предоставляет возможность раздельного управления прозрачностью изображения исходного узла и его отражения. Кроме того, допускается задавать расстояние между узлом и его отражением, а также отражать лишь часть исходного узла. Для создания отражения с заданными свойствами используется следующий конструктор:

```
Reflection(double смещение, double доля,
           double непрозрачность_верх, double непрозрачность_низ)
```

где параметр *смещение* определяет расстояние между нижней частью изображения и его отражением. Предусмотрена возможность отображать лишь часть отраженного узла. Этим управляет параметр *доля*, значения которого должны находиться в пределах от 0 до 1.0. Степенью непрозрачности оригинала и отражения управляют параметры *непрозрачность_верх* и *непрозрачность_низ*, значения которых должны находиться в пределах от 0 до 1.0. Также имеется конструктор по умолчанию, который устанавливает следующие значения параметров: *смещение* — 0, *доля* — 0.75, *непрозрачность_верх* — 0.5 и *непрозрачность_низ* — 0.

Значения параметров могут быть изменены во время выполнения программы. Для изменения параметров непрозрачности предназначены методы setTopOpacity() и setBottomOpacity().

```
final void setTopOpacity(double непрозрачность)
final void setBottomOpacity(double непрозрачность)
```

Значение смещения можно изменить путем вызова метода setTopOffset():

```
final void setTopOffset(double смещение)
```

Задать, какая доля отражения должна отображаться на экране, можно с помощью метода `setFraction()`:

```
final void setFraction(double доля)
```

Перечисленные методы позволяют изменять свойства отражения в процессе выполнения программы.

Преобразования

Преобразования поддерживаются абстрактным классом `Transform`, который находится в пакете `javafx.scene.transform`. В число его подклассов входят классы `Rotate`, `Scale`, `Shear` и `Translate`. Имя каждого класса указывает на его назначение. (У класса `Transform` имеется еще и пятый подкласс — `Affine`, но обычно вам будет достаточно использовать первые четыре подкласса.) К узлу могут быть применены одновременно несколько преобразований. Например, можно повернуть узел и одновременно масштабировать его. Преобразования поддерживаются классом `Node`.

Одним из способов добавления преобразования в узел является его включение в список преобразований, поддерживаемых узлом. Этот список можно получить с помощью метода `getTransforms()`, определяемого классом `Node`. Вот общая форма объявления этого метода:

```
final ObservableList<Transform> getTransforms()
```

Данный метод возвращает ссылку на список преобразований. Чтобы добавить преобразование, достаточно включить его в этот список, вызвав метод `add()`. Список можно очистить, вызвав метод `clear()`. Для удаления конкретного элемента списка можно вызвать метод `remove()`.

В некоторых случаях преобразование можно задать непосредственно, установив одно из свойств класса `Node`. Например, можно задать угол поворота узла вокруг его центра, вызвав метод `setRotate()` с передачей ему требуемого значения угла в качестве параметра. Для масштабирования узла используются методы `setScaleX()` и `setScaleY()`, а для его переноса — методы `setTranslateX()` и `setTranslateY()`. (Преобразования с использованием Z-координаты также могут поддерживаться платформой.) Однако использование списков преобразований обеспечивает большую гибкость, и именно такого подхода мы будем придерживаться далее.

Для демонстрационного примера выбраны классы `Rotate` и `Scale`. (Другие классы преобразований используются аналогичным образом.) Преобразование `Rotate` осуществляет поворот узла на заданный угол вокруг заданной точки. Эти значения могут быть установлены при создании экземпляра `Rotate`. Вот синтаксис объявления одного из конструкторов класса `Rotate`:

```
Rotate(double угол, double x, double y)
```

где параметр *`угол`* определяет угол поворота в градусах. Координаты центра поворота, называемого опорной точкой, определяются параметрами *`x`* и *`y`*.

Можно использовать конструктор по умолчанию, а затем установить параметры поворота уже после создания объекта `Rotate`, что и сделано в демонстрационной программе, представленной в следующем разделе. Для этого используются методы `setAngle()`, `setPivotX()` и `setPivotY()`.

```
final void setAngle(double угол)
final void setPivotX(double x)
final void setPivotY(double y)
```

Как уже отмечалось, параметр *угол* определяет величину угла поворота в градусах, а координаты центра поворота задаются параметрами *x* и *y*. С помощью указанных методов вы сможете повернуть узел в процессе выполнения программы, что позволяет создавать яркие эффекты.

Преобразование `Scale` масштабирует узел в соответствии с заданным коэффициентом масштабирования, благодаря чему можно изменить размеры узла. Класс `Scale` определяет несколько конструкторов. Мы будем использовать конструктор следующего вида:

```
Scale(double коэффициент_ширина, double коэффициент_высота)
```

где параметр *коэффициент_ширина* определяет коэффициент масштабирования для ширины узла, а параметр *коэффициент_высота* — значение коэффициента, применяемое к высоте узла. После создания экземпляра класса `Scale` значения этих параметров могут быть изменены с помощью методов `setX()` и `setY()`.

```
final void setX(double коэффициент_ширина)
final void setY(double коэффициент_высота)
```

С помощью указанных методов вы сможете изменить размеры компонента во время выполнения программы, что может быть использовано для привлечения внимания пользователя.

Демонстрация эффектов и преобразований

Ниже приведена программа, в которой демонстрируется использование эффектов и преобразований. В программе создаются три кнопки — **Повернуть**, **Масштабировать** и **Размыть** — и метка. При нажатии кнопки к ней применяется соответствующий эффект или преобразование. В частности, нажатиям кнопок соответствуют следующие действия: **Повернуть** — поворот кнопки на 15 градусов; **Масштабировать** — изменение размеров кнопки; **Размыть** — нарастающее размытие кнопки. Эффект отражения иллюстрируется с помощью метки. Исследовав текст программы, вы увидите, как легко выполняется настройка внешнего вида графического интерфейса приложения. Вам будет полезно самостоятельно поэкспериментировать с программой, использовав в ней другие преобразования и эффекты и применяя их к другим типам узлов, а не только к кнопкам.

```
// Демонстрация эффектов поворота, масштабирования,
// отражения и размытия компонента
```

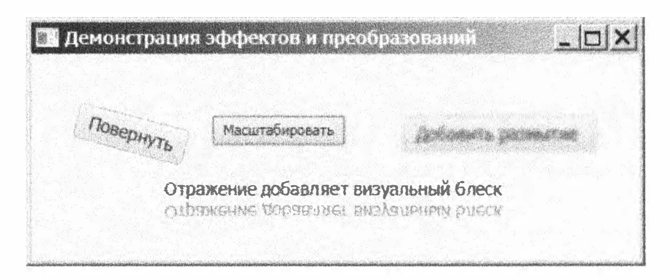
```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.transform.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;

public class EffectsAndTransformsDemo extends Application {

   double angle = 0.0;
   double scaleFactor = 0.4;
   double blurVal = 1.0;

   // Создать начальные объекты преобразований и эффектов
   Reflection reflection = new Reflection();
   BoxBlur blur = new BoxBlur(1.0, 1.0, 1);
   Rotate rotate = new Rotate();
   Scale scale = new Scale(scaleFactor, scaleFactor);

   // Создать кнопки
   Button btnRotate = new Button("Повернуть");
   Button btnBlur = new Button("Размыть");
   Button btnScale = new Button("Масштабировать");

   Label reflect = new Label("Отражение добавляет визуальный блеск");

   public static void main(String[] args) {

      // Запустить приложение JavaFX, вызвав метод launch()
      launch(args);
   }

   // Переопределить метод start()
   public void start(Stage myStage) {

      // Задать заголовок окна приложения
      myStage.setTitle("Демонстрация эффектов и преобразований");

      // Использовать компоновку FlowPane для корневого узла.
      // В данном случае величина вертикального и горизонтального
      // зазоров составляет 20.
      FlowPane rootNode = new FlowPane(20, 20);

      // Центрировать компоненты на сцене
      rootNode.setAlignment(Pos.CENTER);
```

Создание эффектов и преобразований

```
// Создать сцену
Scene myScene = new Scene(rootNode, 300, 120);

// Установить сцену на платформе
myStage.setScene(myScene);

// Добавить поворот в список преобразований для
// кнопки "Повернуть"
btnRotate.getTransforms().add(rotate); ◄—— Добавление вращения к кнопке btnRotate

// Добавить масштабирование в список преобразований
// для кнопки "Масштабировать"
btnScale.getTransforms().add(scale); ◄—— Добавление отражения к кнопке btnScale

// Задать эффект отражения для метки
reflection.setTopOpacity(0.7);
reflection.setBottomOpacity(0.3);
reflect.setEffect(reflection); ◄—— Настройка отражения для метки reflect

// Обработать события действий для кнопки "Повернуть"
btnRotate.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
      // При каждом нажатии кнопки она поворачивается
      // на 15 градусов вокруг центра
      angle += 15.0;

      rotate.setAngle(angle);
      rotate.setPivotX(btnRotate.getWidth()/2);
      rotate.setPivotY(btnRotate.getHeight()/2);
   }
});

// Обработать события действий для кнопки "Масштабировать"
btnScale.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
      // При каждом нажатии кнопки изменяются ее размеры
      scaleFactor += 0.1;
      if(scaleFactor > 2.0) scaleFactor = 0.4;

      scale.setX(scaleFactor);
      scale.setY(scaleFactor);

   }
});

// Обработать события действий для кнопки "Размыть"
btnBlur.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
      // При каждом нажатии кнопки изменяется
      // степень размытия ее изображения
      if(blurVal == 10.0) {
```

```
            blurVal = 1.0;
            btnBlur.setEffect(null); ←——— Удалить размытие с кнопки btnBlur
            btnBlur.setText("Отменить размытие");
         } else {
            blurVal++;
            btnBlur.setEffect(blur); ←——— Добавить размытие к кнопке btnBlur
            btnBlur.setText("Добавить размытие");
         }
         blur.setWidth(blurVal);
         blur.setHeight(blurVal);
      }
   });

   // Добавить метку и кнопки в граф сцены
   rootNode.getChildren().
     addAll(btnRotate, btnScale, btnBlur, reflect);

   // Отобразить платформу вместе с ее сценой
   myStage.show();
  }
}
```

Результат выполнения программы представлен на приведенном ниже рисунке.



Завершая обсуждение эффектов и преобразований, следует отметить, что некоторые из них выглядят особенно выигрышно, когда применяются к текстовым узлам. Класс `Text`, находящийся в пакете `javafx.scene.text`, создает узел, состоящий из текста. Поскольку сам текст в данном случае является узлом, им можно легко манипулировать как целым объектом, применяя к нему всевозможные эффекты и преобразования.

Что дальше

Примите поздравления! Если вы прочитали и проработали материал всех 17 глав, то можете смело считать себя Java-программистом. Конечно, вам предстоит узнать еще немало о самом языке Java, его библиотеках и подсистемах, но вы уже владеете достаточным запасом базовых знаний, который послужит вам надежным фундаментом для приобретения дополнительных знаний и опыта.

В дальнейшем вам, вероятнее всего, придется углубленно изучить следующие темы.

JavaFX и Swing. Обе эти технологии играют важную роль в современном программировании на Java.

Обработка событий.

Классы Java, обеспечивающие работу в сети.

Служебные классы Java, в особенности из библиотеки Collections Framework, которые упрощают решение ряда распространенных задач программирования.

Программный интерфейс Concurrent API, позволяющий повысить надежность и производительность многопоточных приложений.

Технология JavaBeans, предназначенная для создания программных Java-компонентов.

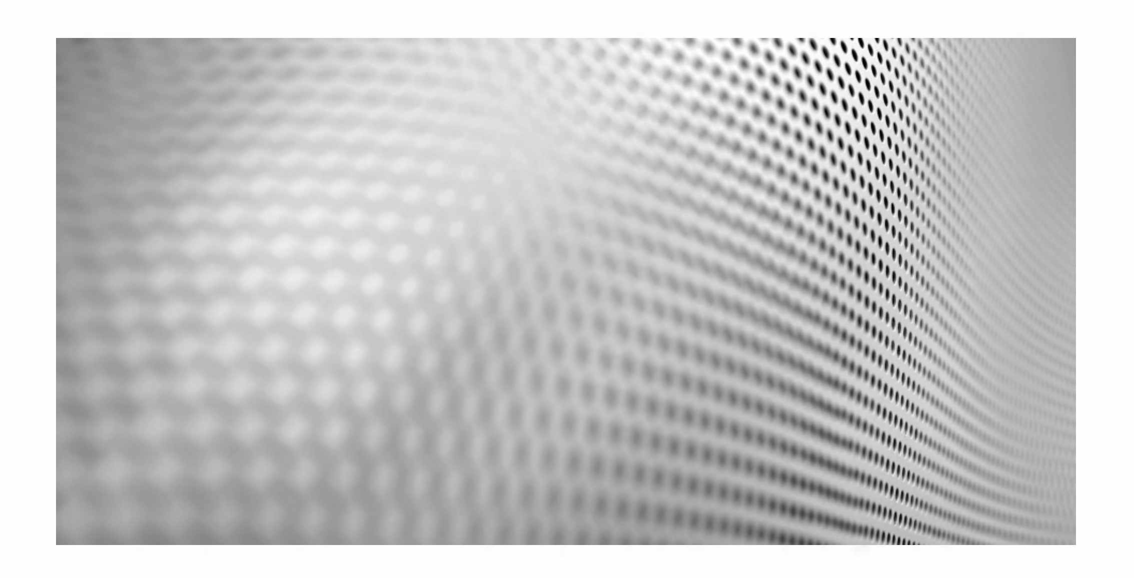
Сервлеты. Если вам придется писать высокопроизводительные веб-приложения, то вы вряд ли сможете обойтись без навыков разработки сервлетов. Сервлеты играют на стороне сервера ту же роль, что и аплеты на стороне клиента.

Для дальнейшего изучения Java рекомендуется прочитать книгу *Java. Полное руководство, 10-е издание*. В ней вы найдете подробные сведения о языке программирования Java и его основных библиотеках, а также сотни примеров программ, демонстрирующих возможности Java.

Вопросы и упражнения для самопроверки

1. Назовите имя пакета верхнего уровня библиотеки JavaFX.
2. Двумя центральными понятиями в JavaFX являются платформа и сцена. Какие классы их инкапсулируют?
3. Граф сцены состоит из ____.
4. Базовым классом для всех узлов служит класс ___________________.
5. Какой класс должны расширять все приложения JavaFX?
6. Какие три метода управляют жизненным циклом приложения JavaFX?
7. В каком из методов, управляющих жизненным циклом, возможно создание платформы приложения?
8. Метод `launch()` вызывается для запуска автономного приложения JavaFX. Верно или неверно?

9. Назовите классы JavaFX, которые реализуют метку и кнопку.
10. Одним из способов прекращения работы автономного приложения JavaFX является вызов метода `Platform.exit()`. Класс `Platform` находится в пакете `javafx.Application`. При вызове метода `exit()` работа программы немедленно прекращается. Учитывая это, измените программу `JavaFXEventDemo`, представленную в данной главе, таким образом, чтобы она отображала две кнопки: **Выполнить** и **Выход**. При нажатии кнопки **Выполнить** программа должна вывести соответствующее сообщение в метке. При нажатии кнопки **Выход** приложение должно завершить свою работу. В обработчиках событий используйте лямбда-выражения.
11. Какой компонент JavaFX реализует флажок?
12. Класс `ListView` — это компонент, который отображает список файлов, находящихся в некотором каталоге локальной файловой системы. Верно или неверно?
13. Преобразуйте Swing-программу для сравнения файлов из упражнения 16.1 в приложение JavaFX. При этом воспользуйтесь предоставляемой в JavaFX возможностью запускать события действий для кнопки программными средствами. Это делается путем вызова метода `fire()` для экземпляра кнопки. К примеру, если имеется экземпляр класса `Button`, который вы назвали `myButton`, то для запуска события необходимо вызвать метод `myButton.fire()`. Воспользуйтесь этим при реализации обработчиков событий для текстовых полей, в которых хранятся имена сравниваемых файлов. В тех случаях, когда пользователь нажимает клавишу <Enter> и при этом фокус ввода находится в одном из указанных текстовых полей, запускайте событие действия для кнопки **Сравнить**. После этого код обработчика событий для кнопки **Сравнить** должен выполнить сравнение файлов.
14. Модифицируйте программу `EffectsAndTransformsDemo` таким образом, чтобы размытие применялось также к кнопке **Повернуть**. Задайте для ширины и высоты области размытия значение `5`, а для счетчика итераций — значение `2`.
15. Самостоятельно поэкспериментируйте с другими эффектами и преобразованиями. Например, попытайтесь использовать эффект свечения и преобразование `Translate`.
16. Продолжайте углублять свои знания в области Java. Оптимальный вариант — изучить базовые пакеты Java, такие как `java.lang`, `java.util` и `java.net`. Напишите тестовые программы, демонстрирующие использование различных классов и интерфейсов из этих пакетов. В общем, для того чтобы стать профессиональным Java-программистом, не существует лучшего способа, чем писать как можно больше программ на Java.



Приложение А

Ответы на вопросы и решения упражнений для самопроверки

Глава 1

1. Что такое байт-код и почему он так важен для интернет-программирования на языке Java?
   Байт-код — это высокооптимизированный набор инструкций, выполняемых под управлением виртуальной машины Java. Использование байт-кода помогает улучшить переносимость и безопасность программ на Java.
2. Каковы три ключевых принципа объектно-ориентированного программирования?
   Инкапсуляция, полиморфизм и наследование.
3. С чего начинается выполнение программы на Java?
   Выполнение программы на Java начинается с метода `main()`.
4. Что такое переменная?
   Переменная — это именованная область памяти. Содержимое переменной может изменяться в процессе выполнения программы.
5. Какое из перечисленных ниже имен переменных недопустимо?
   А. `count`
   Б. `$count`
   В. `count27`
   Г. `67count`
   Недопустимо имя `67count` (пункт *г*). Имя переменной не должно начинаться с цифры.
6. Как создаются однострочные и многострочные комментарии?
   Однострочные комментарии должны начинаться с символов `//`. В данном случае комментариями считаются эти и все последующие символы до конца строки. Многострочные комментарии должны начинаться символами `/*` и заканчиваться символами `*/`.
7. Как выглядит общая форма условной инструкции `if`? Как выглядит общая форма цикла `for`?
   Общая форма инструкции `if` выглядит следующим образом:
   ```
   if (условие) инструкция;
   ```
   Общая форма цикла `for` такова:
   ```
   for(инициализация; условие; итерация) инструкция;
   ```
8. Как создать блок кода?
   Блок кода должен начинаться с символа { и заканчиваться символом }.
9. Сила тяжести на Луне составляет около 17% земной. Напишите программу, которая вычислила бы ваш вес на Луне.
   ```
   /*
      Вычисление веса на Луне.

      Присвойте этому файлу имя Moon.java.
   */
   ```

```
class Moon {
   public static void main(String args[]) {
      double earthweight; // вес на Земле
      double moonweight; // вес на Луне

      earthweight = 165; // вес на Земле в фунтах

      moonweight = earthweight * 0.17;

      System.out.println(earthweight + " земных фунтов
                         эквивалентны " + moonweight +
                         " лунным фунтам.");
   }
}
```

10. Видоизмените программу, созданную в упражнении 1.2, таким образом, чтобы она выводила таблицу перевода дюймов в метры. Выведите значения длины до 12 футов через каждый дюйм. После каждых 12 дюймов выведите пустую строку. (Один метр приблизительно равен 39,37 дюйма.)

```
/*
   Эта программа отображает таблицу преобразования дюймов в метры.

   Присвойте этому файлу имя InchToMeterTable.java.
*/

class InchToMeterTable {
   public static void main(String args[]) {
      double inches, meters;
      int counter;

      counter = 0;
      for(inches = 1; inches <= 144; inches++) {
         meters = inches * 39.37; // преобразовать в метры
         System.out.println(inches + " дюймов равно " +
                            meters + " метров.");

         counter++;
         // Каждая 12-я выводимая строка должна быть пустой
         if(counter == 12) {
            System.out.println();
            counter = 0; // сбросить счетчик строк
         }
      }
   }
}
```

11. Если при вводе кода программы вы допустите опечатку, то какого рода сообщение об ошибке получите?
Сообщение о синтаксической ошибке.

12. Имеет ли значение, с какой именно позиции в строке начинается инструкция?
Не имеет. В Java допускается произвольное форматирование исходного кода.

Глава 2

1. Почему в Java строго определены диапазоны допустимых значений и области действия простых типов?
   Диапазоны допустимых значений и области действия простых типов строго определены в Java для того, чтобы обеспечить переносимость программ с одной платформы на другую.
2. Что собой представляет символьный тип в Java и чем он отличается от символьного типа в ряде других языков программирования?
   Символьный тип задается ключевым словом `char`. В Java для представления символов используется кодировка Unicode, а не ASCII, как во многих других языках программирования.
3. Переменная типа `boolean` может иметь любое значение, поскольку любое ненулевое значение интерпретируется как истинное. Верно или неверно?
   Неверно. Переменная типа `boolean` может иметь лишь значение `true` или `false`.
4. Допустим, результат выполнения программы выглядит следующим образом.
   ```
   Один
   Два
   Три
   ```
   Напишите строку кода с вызовом метода `println()`, где этот результат выводится в виде одной строки.
   ```
   System.out.println("Один\nДва\nТри");
   ```
5. Какая ошибка допущена в следующем фрагменте кода?
   ```
   for(i = 0; i < 10; i++) {
      int sum;

      sum = sum + i;
   }
   System.out.println("Сумма: " + sum);
   ```
   В этом фрагменте кода имеются две грубые ошибки. Во-первых, переменная `sum` заново создается на каждом шаге цикла `for`, следовательно, в промежутке между последовательными итерациями предыдущее значение подсчитываемой суммы не будет сохраняться в этой переменной. И во-вторых, переменная `sum` недоступна за пределами блока кода, в котором она объявлена. Поэтому ссылка на нее при вызове метода `println()` недопустима.
6. Поясните различие между префиксной и постфиксной формами записи операции инкремента.
   Если оператор инкремента предшествует операнду, исполняющая среда Java выполнит операцию до извлечения значения операнда и использования его в остальной части выражения. Если же оператор инкремента следует за операндом, исполняющая среда сначала извлечет значение операнда и лишь затем инкрементирует сам операнд.

7. **Покажите, каким образом укороченный логический оператор И может предотвратить деление на нуль.**

```
if((b != 0) && (val / b)) ...
```

8. **До какого типа повышаются типы `byte` и `short` при вычислении выражения?**

   В выражениях типы `byte` и `short` повышаются до типа `int`.

9. **Когда возникает потребность в явном приведении типов?**

   Явное приведение типов требуется при выполнении преобразований между несовместимыми типами, а также в случае преобразований, сужающих диапазон допустимых значений.

10. **Напишите программу, которая находила бы все простые числа в диапазоне от 2 до 100.**

```
// Нахождение простых чисел в диапазоне от 2 до 100
class Prime {
   public static void main(String args[]) {
      int i, j;
      boolean isprime;

      for(i=2; i < 100; i++) {
         isprime = true;

         // Проверить, делится ли число без остатка
         for(j=2; j <= i/j; j++)
            // Если число делится без остатка, значит, оно не простое
            if((i%j) == 0) isprime = false;

         if(isprime)
            System.out.println(i + " - простое число.");
      }
   }
}
```

11. **Влияют ли лишние скобки на эффективность выполнения программ?**

    Нет, не влияют.

12. **Определяет ли блок кода область действия переменных?**

    Да, определяет.

Глава 3

1. **Напишите программу, которая считывает символы с клавиатуры до тех пор, пока не встретится точка. Предусмотрите в программе счетчик пробелов. Сведения о количестве пробелов должны выводиться в конце программы.**

```
// Подсчет пробелов
class Spaces {
   public static void main(String args[])
   throws java.io.IOException {
```

```
        char ch;
        int spaces = 0;

        System.out.println("Для остановки введите символ точки.");

        do {
           ch = (char) System.in.read();
           if(ch == ' ') spaces++;
        } while(ch != '.');

        System.out.println("Пробелов: " + spaces);
    }
}
```

2. **Каков общий синтаксис многоступенчатой конструкции `if-else-if`?**

```
if(условие)
   инструкция;
else if(условие)
   инструкция;
else if(условие)
   инструкция;
   .
   .
   .
else
   инструкция;
```

3. **Допустим, имеется следующий фрагмент кода.**

```
if(x < 10)
   if(y > 100) {
      if(!done) x = z;
      else y = z;
   }
else System.out.println("ошибка"); // к какой инструкции if относится?
```

С какой из инструкций `if` связана последняя ветвь `else`?

Последняя инструкция `else` соответствует инструкции `if(y > 100)`.

4. **Напишите цикл `for`, в котором перебирались бы значения от 1000 до 0 с шагом 2.**

```
for(int i = 1000; i >= 0; i -= 2) // ...
```

5. **Корректен ли следующий фрагмент кода?**

```
for(int i = 0; i < num; i++)
   sum += i;

count = i;
```

Нет, не корректен. Переменная `i` недоступна за пределами цикла `for`, в котором она объявлена.

6. **Какие действия выполняет инструкция `break`? Опишите оба варианта этой инструкции.**

Инструкция `break` без метки вызывает немедленное завершение текущего цикла или инструкции `switch`. Инструкция `break` с меткой передает управление в конец помеченного блока.

7. Какое сообщение будет выведено после выполнения инструкции `break` в приведенном ниже фрагменте кода?

```
for(i = 0; i < 10; i++) {
   while(running) {
      if(x<y) break;
      // ...
   }
   System.out.println("После while");
}
System.out.println("После for");
```

После выполнения инструкции `break` будет выведено сообщение `"После while"`.

8. Что будет выведено на экран в результате выполнения следующего фрагмента кода?

```
for(int i = 0; i<10; i++) {
   System.out.print(i + " ");
   if((i%2) == 0) continue;
   System.out.println();
}
```

На экране появится следующий результат.

```
0 1
2 3
4 5
6 7
8 9
```

9. Итерационное выражение для цикла `for` не обязательно должно изменять переменную цикла на фиксированную величину. Эта переменная может иметь произвольные значения. Напишите программу, использующую цикл `for` для вывода чисел в геометрической прогрессии: 1, 2, 4, 8, 16, 32 и т.д.

```
/*
   Использование цикла for для формирования
   геометрической прогрессии:

   1, 2, 4, 8, 16, 32...
*/
class Progress {
   public static void main(String args[]) {

      for(int i = 1; i < 100; i += i)
         System.out.print(i + " ");
   }
}
```

10. Коды ASCII-символов нижнего регистра отличаются от кодов соответствующих символов верхнего регистра на величину 32. Следовательно, для преобразования строчной буквы в прописную нужно уменьшить ее код на 32. Используйте это обстоятельство для написания программы, читающей символы с клавиатуры. При выводе результатов данная программа должна преобразовывать строчные буквы в прописные, а прописные — в строчные. Остальные символы не должны меняться. Работа программы должна завершаться после того, как пользователь введет с клавиатуры точку. И наконец, сделайте так, чтобы программа отображала количество символов, для которых был изменен регистр.

```
// Смена регистра символов
class CaseChg {
   public static void main(String args[]) throws java.io.IOException {
      char ch;
      int changes = 0;

      System.out.println("Для остановки введите символ точки.");

      do {
         ch = (char) System.in.read();
         if(ch >= 'a' & ch <= 'z') {
            ch -= 32;
            changes++;
            System.out.println(ch);
         }
         else if(ch >= 'A' & ch <= 'Z') {
            ch += 32;
            changes++;
            System.out.println(ch);
         }
      } while(ch != '.');
      System.out.println("Изменение регистра: " + changes);
   }
}
```

11. Что такое бесконечный цикл?
Бесконечным называется цикл, выполнение которого никогда не прекращается.

12. Должна ли метка, используемая в инструкции `break`, быть определена в блоке кода, содержащем эту инструкцию?
Да, должна.

Глава 4

1. Чем отличается класс от объекта?
Класс — это абстрактное логическое описание структуры и поведения объекта, тогда как объект — это конкретный экземпляр класса.

2. Как определяется класс?
   Класс определяется с помощью ключевого слова `class`. В инструкции `class` указываются код и данные, составляющие класс.
3. Собственную копию чего содержит каждый объект?
   Каждый объект класса содержит собственную копию переменных экземпляра этого класса.
4. Покажите, как объявить объект `counter` класса `MyCounter`, используя две отдельные инструкции.
   ```
   MyCounter counter;
   counter = new MyCounter();
   ```
5. Как должен быть объявлен метод `myMeth()`, имеющий два параметра, `a` и `b`, типа `int` и возвращающий значение типа `double`?
   ```
   double myMeth(int a, int b) { // ...
   ```
6. Как должно завершаться выполнение метода, возвращающего некоторое значение?
   Для завершения метода служит инструкция `return`. Она же передает возвращаемое значение вызывающему блоку программы.
7. Каким должно быть имя конструктора?
   Имя конструктора должно совпадать с именем класса.
8. Какие действия выполняет оператор `new`?
   Оператор `new` выделяет память для объекта и выполняет его инициализацию, используя конструктор.
9. Что такое сборка мусора и для чего она нужна?
   Сборка мусора — это механизм удаления неиспользуемых объектов для повторного использования освобождаемой памяти.
10. Что означает ключевое слово `this`?
    Ключевое слово `this` означает ссылку на объект, для которого вызывается метод. Эта ссылка автоматически передается методу.
11. Может ли конструктор иметь один или несколько параметров?
    Да, может.
12. Если метод не возвращает значения, то как следует объявить тип этого метода?
    Как `void`.

Глава 5

1. Продемонстрируйте два способа объявления одномерного массива, состоящего из 12 элементов типа `double`.
   ```
   double x[] = new double[12];
   double[] x = new double[12];
   ```
2. Покажите, как инициализировать одномерный массив целочисленными значениями от 1 до 5.
   ```
   int x[] = { 1, 2, 3, 4, 5 };
   ```

3. **Напишите программу, в которой массив используется для нахождения среднего арифметического десяти значений типа `double`. Используйте любые десять чисел.**

```
// Среднее арифметическое 10 значений типа double
class Avg {
   public static void main(String args[]) {
      double nums[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8,
                        9.9, 10.1 };
      double sum = 0;

      for(int i=0; i < nums.length; i++)
         sum += nums[i];

      System.out.println("Среднее значение: " + sum / nums.length);
   }
}
```

4. **Измените программу, созданную в упражнении 5.1, таким образом, чтобы она сортировала массив строк. Продемонстрируйте ее работоспособность.**

```
// Демонстрация алгоритма пузырьковой сортировки строк
class StrBubble {
   public static void main(String args[]) {
      String strs[] = { "this", "is", "a", "test",
                        "of", "a", "string", "sort"
                      };
      int a, b;
      String t;
      int size;

      size = strs.length; // количество сортируемых элементов

      // Отображение исходного массива
      System.out.print("Исходный массив:");
      for(int i=0; i < size; i++)
         System.out.print(" " + strs[i]);
      System.out.println();

      // Пузырьковая сортировка строк
      for(a=1; a < size; a++)
         for(b=size-1; b >= a; b--) {
            // поменять элементы местами при нарушении
            // порядка их следования
            if(strs[b-1].compareTo(strs[b]) > 0) {
               t = strs[b-1];
               strs[b-1] = strs[b];
               strs[b] = t;
            }
         }

      // Отображение отсортированного массива
      System.out.print("Отсортированный массив:");
```

```
        for(int i=0; i < size; i++)
            System.out.print(" " + strs[i]);
        System.out.println();
    }
}
```

5. **В чем отличие методов `indexOf()` и `lastIndexOf()` класса `String`?**
   Метод `indexOf()` находит первое вхождение указанной подстроки, а метод `lastIndexOf()` — ее последнее вхождение в текущей строке.
6. **Все символьные строки являются объектами типа `String`. Покажите, как вызываются методы `length()` и `charAt()` для строкового литерала `"Мне нравится Java"`.**
   Как ни странно, приведенный ниже вызов метода `length()` вполне допустим.

   ```
   System.out.println("Мне нравится Java".length());
   ```

   В результате на экран выводится значение `17`. Аналогичным образом вызывается и метод `charAt()`.
7. **Расширьте класс `Encode` таким образом, чтобы в качестве ключа шифрования использовалась строка из восьми символов.**

```
// Улучшенный вариант программы шифрования сообщений
// с помощью побитовой операции исключающего ИЛИ
class Encode {
    public static void main(String args[]) {
        String msg = "This is a test";
        String encmsg = "";
        String decmsg = "";
        String key = "abcdefgi";
        int j;

        System.out.print("Исходное сообщение: ");
        System.out.println(msg);

        // Шифрование сообщение
        j = 0;
        for(int i=0; i < msg.length(); i++) {
            encmsg = encmsg + (char) (msg.charAt(i) ^ key.charAt(j));
            j++;
            if(j==8) j = 0;
        }

        System.out.print("Зашифрованное сообщение: ");
        System.out.println(encmsg);

        // Дешифровка сообщения
        j = 0;
        for(int i=0; i < msg.length(); i++) {
            decmsg = decmsg + (char) (encmsg.charAt(i) ^ key.charAt(j));
            j++;
```

```
            if(j==8) j = 0;
        }

        System.out.print("Дешифрованное сообщение: ");
        System.out.println(decmsg);

    }
}
```

8. **Можно ли применять побитовые операции к значениям типа double?**
   Нет, нельзя.
9. **Перепишите приведенную ниже последовательность инструкций, воспользовавшись оператором ?.**

   ```
   if(x < 0) y = 10;
   else y = 20;
   ```

   Ответ:

   ```
   y = x < 0 ? 10 : 20;
   ```

10. **В приведенном ниже фрагменте кода содержится знак &. Какой операции он соответствует: побитовой или логической? Обоснуйте свой ответ.**

    ```
    boolean a, b;
    // ...
    if(a & b) ...
    ```

    Это логическая операция, поскольку оба операнда относятся к типу boolean.
11. **Является ли ошибкой превышение верхней границы массива?**
    Да.
    Является ли ошибкой использование отрицательных значений для доступа к элементам массива?
    Да. Значения индексов массива начинаются с нуля.
12. **Как обозначается операция сдвига вправо без знака?**
    >>>
13. **Перепишите рассмотренный ранее класс MinMax таким образом, чтобы в нем использовался цикл типа for-each.**

    ```
    // Поиск минимального и максимального значений в массиве
    class MinMax {
       public static void main(String args[]) {
          int nums[] = new int[10];
          int min, max;

          nums[0] = 99;
          nums[1] = -10;
          nums[2] = 100123;
          nums[3] = 18;
          nums[4] = -978;
          nums[5] = 5623;
    ```

```
        nums[6] = 463;
        nums[7] = -9;
        nums[8] = 287;
        nums[9] = 49;

        min = max = nums[0];
        for(int v : nums) {
           if(v < min) min = v;
           if(v > max) max = v;
        }
        System.out.println("min и max: " + min + " " + max);
    }
}
```

14. **В упражнении 5.1 была реализована пузырьковая сортировка. Можно ли в программе из этого примера заменить обычный цикл `for` циклом типа `for-each`? Если нельзя, то почему?**

    Циклы `for`, выполняющие сортировку в классе `Bubble`, нельзя преобразовать в вариант `for-each`. Что касается внешнего цикла, то текущее значение его переменной используется во внутреннем цикле. А в случае внутреннего цикла для перестановки неупорядоченных элементов требуется присваивать значения элементам массива, что недопустимо в цикле типа `for-each`.

15. **Можно ли управлять инструкцией `switch` с помощью объектов типа `String`?**

    Можно, начиная с версии JDK 7.

Глава 6

1. **Предположим, имеется следующий фрагмент кода.**

    ```
    class X {
       private int count;
    ```

    Исходя из этого, допустим ли следующий код?

    ```
    class Y {
       public static void main(String args[]) {
          X ob = new X();

          ob.count = 10;
    ```

    Нет. Закрытый (`private`) член недоступен за пределами своего класса.

2. **Модификатор доступа должен ______________ объявлению члена класса.**

    предшествовать

3. **Помимо очереди, в программах часто используется структура данных, которая называется стеком. Обращение к стеку осуществляется по принципу “первым пришел — последним обслужен“. Стек можно сравнить со стопкой тарелок, стоящих на столе. Последней берется тарелка, поставленная**

на стол первой. Создайте класс `Stack`, реализующий стек для хранения символов. Используйте методы `push()` и `pop()` для манипулирования содержимым стека. Пользователь класса `Stack` должен иметь возможность задавать размер стека при его создании. Все члены класса `Stack`, кроме методов `push()` и `pop()`, должны быть объявлены как `private`. (Подсказка: в качестве заготовки можете воспользоваться классом `Queue`, изменив в нем лишь способ доступа к данным.)

```
// Класс, реализующий стек для хранения символов
class Stack {
   private char stck[]; // массив для хранения элементов стека
   private int tos;     // вершина стека

   // Создать пустой стек заданного размера
   Stack(int size) {
      stck = new char[size]; // выделить память для стека
      tos = 0;
   }

   // Создать один стек на основе другого стека
   Stack(Stack ob) {
      tos = ob.tos;
      stck = new char[ob.stck.length];

      // Скопировать элементы
      for(int i=0; i < tos; i++)
         stck[i] = ob.stck[i];
      }

   // Создать стек с начальными значениями
   Stack(char a[]) {
      stck = new char[a.length];

      for(int i = 0; i < a.length; i++) {
         push(a[i]);
      }
   }

   // Поместить символ в стек
   void push(char ch) {
      if(tos==stck.length) {
         System.out.println(" -- Стек заполнен");
         return;
      }

      stck[tos] = ch;
      tos++;
   }

   // Извлечь символ из стека
   char pop() {
      if(tos==0) {
```

```
            System.out.println(" -- Стек пуст");
            return (char) 0;
        }

        tos--;
        return stck[tos];
    }
}

// Демонстрация использования класса Stack
class SDemo {
    public static void main(String args[]) {
        // создать пустой стек на 10 элементов
        Stack stk1 = new Stack(10);

        char name[] = {'T', 'o', 'm'};

        // создать стек из массива
        Stack stk2 = new Stack(name);

        char ch;
        int i;

        // поместить символы в стек stk1
        for(i=0; i < 10; i++)
            stk1.push((char) ('A' + i));

        // создать один стек из другого стека
        Stack stk3 = new Stack(stk1);

        // отобразить стеки
        System.out.print("Содержимое stk1: ");
        for(i=0; i < 10; i++) {
            ch = stk1.pop();
            System.out.print(ch);
        }

        System.out.println("\n");

        System.out.print("Содержимое stk2: ");
        for(i=0; i < 3; i++) {
            ch = stk2.pop();
            System.out.print(ch);
        }

        System.out.println("\n");
        System.out.print("Содержимое stk3: ");
        for(i=0; i < 10; i++) {
            ch = stk3.pop();
            System.out.print(ch);
        }
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Содержимое stk1: JIHGFEDCBA
Содержимое stk2: moT
Содержимое stk3: JIHGFEDCBA
```

4. Предположим, имеется следующий класс:

```
class Test {
   int a;
   Test(int i) { a = i; }
}
```

Напишите метод `swap()`, реализующий обмен содержимым между двумя объектами типа `Test`, на которые ссылаются две переменные данного типа.

```
void swap(Test ob1, Test ob2) {
   int t;

   t = ob1.a;
   ob1.a = ob2.a;
   ob2.a = t;
}
```

5. Правильно ли написан следующий фрагмент кода?

```
class X {
   int meth(int a, int b) { ... }
   String meth(int a, int b) { ... }
```

Нет, неправильно. Перегружаемые методы могут возвращать значения разного типа, но это не играет никакой роли при разрешении ссылок на перегруженные версии. Перегружаемые методы должны иметь разные списки параметров.

6. Напишите рекурсивный метод, отображающий строку задом наперед.

```
// Отображение символов строки в обратном порядке
// с помощью рекурсии
class Backwards {
   String str;

   Backwards(String s) {
      str = s;
   }

   void backward(int idx) {
      if(idx != str.length()-1) backward(idx+1);

      System.out.print(str.charAt(idx));
   }
}

class BWDemo {
   public static void main(String args[]) {
      Backwards s = new Backwards("Это тест");
```

```
        s.backward(0);
    }
}
```

7. Допустим, все объекты класса должны совместно использовать одну и ту же переменную. Как объявить такую переменную?

   Переменная, предназначенная для совместного использования, должна быть объявлена как `static`.

8. Для чего может понадобиться статический блок?

   Статический блок служит для выполнения любых инициализирующих действий в классе до создания конкретных объектов.

9. Что такое внутренний класс?

   Внутренний класс — это нестатический вложенный класс.

10. Допустим, требуется член класса, к которому могут обращаться только другие члены этого же класса. Какой модификатор доступа следует использовать в его объявлении?

    Модификатор доступа `private`.

11. Имя метода и список его параметров вместе составляют ____________ метода.

    сигнатуру

12. Если методу передается значение типа `int`, то в этом случае используется передача параметра по ____________.

    значению

13. Создайте метод `sum()`, имеющий список аргументов переменной длины и предназначенный для суммирования передаваемых ему значений типа `int`. Метод должен возвращать результат суммирования. Продемонстрируйте работу этого метода.

    Существует множество вариантов решения данной задачи. Ниже представлен один из них.

```
class SumIt {
    int sum(int ... n) {
        int result = 0;

        for(int i = 0; i < n.length; i++)
            result += n[i];

        return result;
    }
}

class SumDemo {
    public static void main(String args[]) {

        SumIt siObj = new SumIt();

        int total = siObj.sum(1, 2, 3);
        System.out.println("Сумма: " + total);
```

```
        total = siObj.sum(1, 2, 3, 4, 5);
        System.out.println("Сумма: " + total);
    }
}
```

14. Можно ли перегружать методы с переменным числом аргументов?
Да, можно.

15. Приведите пример вызова перегруженного метода с переменным числом аргументов, демонстрирующий возникновение неоднозначности.
Ниже приведен один из вариантов перегружаемого метода с переменным числом аргументов, при вызове которого возникает неоднозначность.

```
double myMeth(double ... v ) { // ...
double myMeth(double d, double ... v) { // ...
```

Если попытаться вызвать метод `myMeth()` с одним аргументом следующим образом:

```
myMeth(1.1);
```

то компилятор не сможет определить, какой именно метод вызывается.

Глава 7

1. Имеет ли суперкласс доступ к членам подкласса?
Нет, не имеет. Суперклассу ничего не известно о существовании подклассов.
Имеет ли подкласс доступ к членам суперкласса?
Подклассы действительно могут обращаться ко всем членам суперкласса, кроме тех, которые объявлены как закрытые (`private`).

2. Создайте подкласс `Circle`, производный от класса `TwoDShape`. В нем должен быть определен метод `area()`, вычисляющий площадь круга, а также конструктор с ключевым словом `super` для инициализации членов, унаследованных от класса `TwoDShape`.

```
// Подкласс для окружностей, производный от класса TwoDShape
class Circle extends TwoDShape {
    // Конструктор по умолчанию
    Circle() {
        super();
    }

    // Конструктор класса Circle
    Circle(double x) {
        super(x, "circle"); // вызвать конструктор суперкласса
    }

    // Создать новый объект из имеющегося объекта
    Circle(Circle ob) {
        super(ob); // передать объект конструктору класса TwoDShape
    }
```

```
    double area() {
        return (getWidth() / 2) * (getWidth() / 2) * 3.1416;
    }
}
```

3. Как предотвратить обращение к членам суперкласса из подкласса?
   Чтобы предотвратить доступ к членам суперкласса из подкласса, эти члены следует объявить как закрытые (`private`).
4. Опишите назначение и два варианта использования ключевого слова `super`.
   Ключевое слово `super` используется в двух случаях. Во-первых, с его помощью вызывается конструктор суперкласса. В этом случае общая форма вызова имеет следующий вид:

   super(*список_параметров*);

   И во-вторых, это ключевое слово обеспечивает доступ к членам суперкласса. Ниже приведена общая форма такого доступа:

   super.*член_класса*
5. Допустим, имеется следующая иерархия классов:

```
class Alpha { ...

class Beta extends Alpha { ...

Class Gamma extends Beta { ...
```

   В каком порядке вызываются конструкторы этих классов при создании объекта класса `Gamma`?
   Конструкторы всегда вызываются в порядке наследования. Таким образом, при создании экземпляра класса `Gamma` сначала будет вызван конструктор `Alpha`, затем `Beta` и наконец `Gamma`.
6. Переменная ссылки на суперкласс может указывать на объект подкласса. Объясните, почему это важно и как это связано с переопределением методов?
   Когда переопределяемый метод вызывается по ссылке на суперкласс, его вариант определяется по типу объекта, на который делается ссылка.
7. Что такое абстрактный класс?
   Абстрактным называется такой класс, который содержит хотя бы один абстрактный метод.
8. Как предотвратить переопределение метода и наследование класса?
   Для того чтобы метод нельзя было переопределить, его нужно объявить как `final`. И для того чтобы предотвратить наследование от класса, его следует объявить как `final`.
9. Объясните, каким образом механизмы наследования, переопределения методов и абстрактные классы используются для поддержки полиморфизма.

Наследование, переопределение методов и абстрактные классы поддерживают полиморфизм и позволяют создать обобщенную структуру, реализуемую различными классами. Так, абстрактный класс определяет согласованный интерфейс, общий для всех реализующих его классов. Такой подход соответствует принципу "один интерфейс — множество методов".

10. Какой класс является суперклассом для всех остальных классов?
    Класс `Object`.
11. Класс, который содержит хотя бы один абстрактный метод, должен быть объявлен абстрактным. Верно или не верно?
    Верно.
12. Какое ключевое слово следует использовать для создания именованной константы?
    Ключевое слово `final`.

Глава 8

1. Используя код, созданный в упражнении 8.1, поместите в пакет `qpack` интерфейс `ICharQ` и все три реализующих его класса. Оставив класс `IQDemo` в пакете, используемом по умолчанию, покажите, как импортировать и использовать классы из пакета `qpack`.
   Для того чтобы включить интерфейс `ICharQ` и реализующие его классы в пакет `qpack`, следует поместить каждый из них в отдельный файл, объявить все классы, реализующие данный интерфейс, как `public`, а в начале каждого файла указать следующую инструкцию:
   ```
   package qpack;
   ```
   После этого можно использовать пакет `qpack`, добавив в интерфейс `IQDemo` следующую инструкцию `import`:
   ```
   import qpack.*;
   ```
2. Что такое пространство имен? Почему так важна возможность его разделения на отдельные области в Java?
   Пространство имен — это область объявлений. Разделяя пространство имен на отдельные области, можно предотвратить конфликты имен.
3. Содержимое пакетов хранится в _______________.
   каталогах
4. В чем отличие доступа, определяемого ключевым словом `protected`, от доступа по умолчанию?
   Член класса с доступом типа `protected` может быть использован в пределах текущего пакета, а также в подклассах данного класса, относящихся к любому пакету.
   Член класса с доступом по умолчанию может быть использован только в пределах текущего пакета.

5. Допустим, классы, содержащиеся в одном пакете, требуется использовать в другом пакете. Какими двумя способами можно этого добиться?
   Для того чтобы воспользоваться членом пакета, нужно указать его полное имя или же импортировать этот член с помощью инструкции `import`.
6. "Один интерфейс — множество методов" — таков главный принцип Java. Какое языковое средство лучше всего демонстрирует этот принцип?
   Этот принцип объектно-ориентированного программирования лучше всего демонстрирует интерфейс.
7. Сколько классов могут реализовать один и тот же интерфейс? Сколько интерфейсов может реализовать класс?
   Один интерфейс может быть реализован любым количеством классов. Класс может реализовать произвольное число интерфейсов.
8. Может ли один интерфейс наследовать другой?
   Да, может. Механизм наследования распространяется и на интерфейсы.
9. Создайте интерфейс для класса `Vehicle`, рассмотренного в главе 7, назвав его `IVehicle`.

```
interface IVehicle {

   // Вернуть дальность поездки транспортного средства
   int range();

   // Вычислить объем топлива, требующегося
   // для прохождения заданного пути
   double fuelneeded(int miles);

   // Методы доступа к переменным экземпляра
   int getPassengers();
   void setPassengers(int p);
   int getFuelcap();
   void setFuelcap(int f);
   int getMpg();
   void setMpg(int m);
}
```

10. Переменные, объявленные в интерфейсе, неявно имеют модификаторы `static` и `final`. Какие преимущества это дает?
    Переменные, объявленные в интерфейсе, могут использоваться в качестве именованных констант, общих для всех файлов программы. Доступ к ним обеспечивается путем импорта того интерфейса, в котором они объявлены.
11. Пакет по сути является контейнером для классов. Верно или не верно?
    Верно.
12. Какой стандартный пакет автоматически импортируется в любую программу на Java?
    Пакет `java.lang`.
13. Какое ключевое слово используется для объявления в интерфейсе метода по умолчанию?
    Слово `default`.

14. Допускается ли, начиная с JDK 8, определение статического метода интерфейса?
Да.
15. Предположим, что интерфейс ICharQ, представленный в упражнении 8.1, получил широкое распространение в течение нескольких лет. В какой-то момент вы решили добавить в него метод reset(), который будет использоваться для сброса очереди в ее исходное пустое состояние. Как это можно осуществить, не нарушая работоспособность существующего кода, в случае использования комплекта JDK 8 или выше?
Для этого необходимо использовать метод по умолчанию. Вы не можете знать, как сбрасывать каждую реализацию очереди, поэтому стандартная реализация метода reset() должна сообщать об ошибке, указывающей на отсутствие реализации. (Лучше всего генерировать исключение; см. следующую главу.) К счастью, поскольку отсутствует существующий код, предполагающий наличие у интерфейса ICharQ метода reset(), в существующем коде подобная ошибка не возникнет, и его работоспособность не будет нарушена.
16. Как можно вызвать статический метод интерфейса?
Статический метод вызывается с указанием имени интерфейса, после которого ставится разделитель-точка.
17. Может ли интерфейс включать закрытый (private) метод?
Может, начиная с версии JDK 9.

Глава 9

1. Какой класс находится на вершине иерархии исключений?
Класс Throwable.
2. Объясните вкратце, как используются ключевые слова try и catch?
Ключевые слова try и catch используются совместно. Инструкции программы, в которых вы хотите отслеживать исключения, помещаются в блок try. Перехват и обработка исключений осуществляются в блоке catch.
3. Какая ошибка допущена в приведенном ниже фрагменте кода?
```
// ...
vals[18] = 10;
catch (ArrayIndexOutOfBoundsException exc) {
   // обработать ошибку
}
```
Блоку catch не предшествует блок try.
4. Что произойдет, если исключение не будет перехвачено?
Если исключение не будет перехвачено, то произойдет аварийное завершение программы.
5. Какая ошибка допущена в приведенном ниже фрагменте кода?

```
class A extends Exception { ...

class B extends A { ...

// ...

try {
   // ...
}
catch (A exc) { ... }
catch (B exc) { ... }
```

В данном фрагменте кода инструкция catch для суперкласса предшествует инструкции catch для подкласса. А поскольку инструкция catch для суперкласса будет перехватывать также исключения, относящиеся к подклассу, то в программе окажется код, которому никогда не будет передано управление.

6. Может ли внутренний блок catch повторно сгенерировать исключение, которое будет обработано во внешнем блоке catch?
Да, исключения могут генерироваться повторно.
7. Блок finally — последний фрагмент кода, выполняемый перед завершением программы. Верно или неверно? Обоснуйте свой ответ.
Неверно. Блок finally выполняется по завершении блока try.
8. Исключения какого типа необходимо явно объявлять с помощью инструкции throws, включаемой в объявление метода?
С помощью инструкции throws объявляются все исключения, кроме RuntimeException и Error.
9. Какая ошибка допущена в приведенном ниже фрагменте кода?

```
class MyClass { // ... }
// ...
throw new MyClass();
```

Класс MyClass не является производным от класса Throwable. С помощью инструкции throw могут генерироваться лишь те исключения, которые являются подклассами, производными от Throwable.

10. Отвечая на вопрос 3 в конце главы 6, вы создали класс Stack. Добавьте в него пользовательские исключения, чтобы программа нужным образом реагировала на попытку поместить элемент в заполненный стек или извлечь элемент из пустого стека.

```
// Исключение, возникающее при переполнении стека
class StackFullException extends Exception {
   int size;

   StackFullException(int s) { size = s; }

   public String toString() {
      return "\nСтек заполнен. Максимальный размер стека: " + size;
   }
}
```

```
// Исключение, возникающее при обращении к пустому стеку
class StackEmptyException extends Exception {

   public String toString() {
      return "\nСтек пуст.";
   }
}

// Класс, реализующий стек для хранения символов
class Stack {
   private char stck[]; // массив для хранения элементов стека
   private int tos;     // вершина стека

   // Создать пустой стек заданного размера
   Stack(int size) {
      stck = new char[size]; // выделить память для стека
      tos = 0;
   }

   // Создать один стек на основе другого стека
   Stack(Stack ob) {
      tos = ob.tos;
      stck = new char[ob.stck.length];

      // Скопировать элементы
      for(int i=0; i < tos; i++)
         stck[i] = ob.stck[i];
   }

   // Создать стек с начальными значениями
   Stack(char a[]) {
      stck = new char[a.length];

      for(int i = 0; i < a.length; i++) {
         try {
            push(a[i]);
         }
         catch(StackFullException exc) {
            System.out.println(exc);
         }
      }
   }

   // Поместить символ в стек
   void push(char ch) throws StackFullException {
      if(tos==stck.length)
         throw new StackFullException(stck.length);

      stck[tos] = ch;
      tos++;
   }

   // Извлечь символ из стека
```

```
    char pop() throws StackEmptyException {
        if(tos==0)
            throw new StackEmptyException();

        tos--;
        return stck[tos];
    }
}
```

11. **Назовите три причины, по которым могут генерироваться исключения.**
    Исключение может быть сгенерировано в результате ошибки в виртуальной машине Java, ошибки в программе или явным образом с помощью инструкции `throw`.
12. **Назовите два подкласса, производных непосредственно от класса `Throwable`.**
    Классы `Error` и `Exception`.
13. **Что такое групповой перехват исключений?**
    Групповым называется такой перехват, который позволяет перехватывать два и более исключения одним блоком `catch`.
14. **Следует ли перехватывать в программе исключения типа `Error`?**
    Нет, не следует.

Глава 10

1. **Для чего в Java определены как байтовые, так и символьные потоки?**
   Первоначально в Java были определены только байтовые потоки. Они особенно удобны для ввода-вывода двоичных данных и поддерживают произвольный доступ к файлам. Символьные потоки оптимизированы для использования кодировки Unicode.
2. **Как известно, консольные операции ввода-вывода осуществляются в текстовом виде. Почему же в Java для этой цели используются байтовые потоки?**
   Стандартные потоки ввода-вывода `System.in`, `System.out` и `System.err` были определены в Java до появления символьных потоков.
3. **Как открыть файл для чтения байтов?**
   Ниже приведен один из способов открытия файла для чтения данных типа `byte`.
   ```
   FileInputStream fin = new FileInputStream("test");
   ```
4. **Как открыть файл для чтения символов?**
   Ниже приведен один из способов открытия файла для чтения символов.
   ```
   FileReader fr = new FileReader("test");
   ```
5. **Как открыть файл для выполнения операций ввода-вывода с произвольным доступом?**

Ниже приведен один из способов открытия файла для выполнения операций ввода-вывода с произвольным доступом.

```
randfile = new RandomAccessFile("test", "rw");
```

6. **Как преобразовать числовую строку "123.23" в ее двоичный эквивалент?**
   Для того чтобы преобразовать числовую строку в ее двоичный эквивалент, следует воспользоваться одним из методов синтаксического разбора, определенных в классах оболочек типов, например `Integer` или `Double`.
7. **Напишите программу для копирования текстовых файлов. Видоизмените ее таким образом, чтобы все пробелы заменялись дефисами. Используйте классы, представляющие байтовые потоки, а также традиционный способ закрытия файла явным вызовом метода `close()`.**

```
/*
   Копирование текстового файла с заменой пробелов дефисами.

   В этой версии программы используются байтовые потоки.

   Для того чтобы воспользоваться этой программой, укажите
   в командной строке имена исходного и целевого файлов. Например:

   java Hyphen source target
*/

import java.io.*;

class Hyphen {
   public static void main(String args[])
   {
      int i;
      FileInputStream fin = null;
      FileOutputStream fout = null;

      // Сначала проверить, указаны ли имена обоих файлов
      if(args.length !=2 ) {
         System.out.println("Использование: Hyphen откуда куда");
         return;
      }

      // Скопировать файл и заменить в нем пробелы дефисами
      try {
         fin = new FileInputStream(args[0]);
         fout = new FileOutputStream(args[1]);

         do {
            i = fin.read();

            // преобразовать пробел в дефис
            if((char)i == ' ') i = '-';

            if(i != -1) fout.write(i);
         } while(i != -1);
```

```
        } catch(IOException exc) {
          System.out.println("Ошибка ввода-вывода: " + exc);
        } finally {
          try {
            if(fin != null) fin.close();
          } catch(IOException exc) {
            System.out.println("Ошибка при закрытии входного файла.");
          }

          try {
            if(fin != null) fout.close();
          } catch(IOException exc) {
            System.out.println("Ошибка при закрытии выходного
файла.");
          }
        }
      }
    }
```

8. **Перепишите программу, созданную в предыдущем пункте, таким образом, чтобы в ней использовались классы, представляющие символьные потоки. На этот раз воспользуйтесь инструкцией `try` с ресурсами для автоматического закрытия файла.**

```
/*
   Копирование текстового файла с заменой пробелов дефисами.

   В этой версии программы используются символьные потоки.

   Для того чтобы воспользоваться этой программой, укажите
   в командной строке имена исходного и целевого файлов. Например:

   java Hyphen2 source target

*/

import java.io.*;

class Hyphen2 {
   public static void main(String args[]) throws IOException
   {
      int i;

      // Сначала проверить, указаны ли имена обоих файлов
      if(args.length !=2 ) {
         System.out.println("Использование: Hyphen2 откуда куда");
         return;
      }

      // Скопировать файл и заменить в нем пробелы дефисами,
      // используя инструкцию try с ресурсами
      try (FileReader fin = new FileReader(args[0]);
           FileWriter fout = new FileWriter(args[1]))
      {
```

```
            do {
               i = fin.read();

               // преобразовать пробел в дефис
               if((char)i == ' ') i = '-';

               if(i != -1) fout.write(i);
            } while(i != -1);
         } catch(IOException exc) {
            System.out.println("Ошибка ввода-вывода: " + exc);
         }
      }
   }
```

9. **К какому типу относится поток `System.in`?**
   К типу `InputStream`.
10. **Какое значение возвращает метод `read()` из класса `InputStream` по достижении конца потока?**
   `-1`.
11. **Поток какого типа используется для чтения двоичных данных?**
   Поток типа `DataInputStream`.
12. **Классы `Reader` и `Writer` находятся на вершине иерархии классов ______________________.**
   символьного ввода-вывода
13. **Инструкция `try` с ресурсами служит для ______________________.**
   автоматического управления ресурсами
14. **Верно ли следующее утверждение: "Если для закрытия файла применяется традиционный способ, то лучше всего делать это в блоке `finally`"?**
   Верно.

Глава 11

1. **Каким образом имеющиеся в Java средства многопоточного программирования обеспечивают создание более эффективных программ?**
   Средства многопоточного программирования дают возможность использовать периоды простоя, возникающие практически в любой программе. Когда операции в одном потоке по каким-то причинам приостановлены, выполняются другие потоки. В многоядерных системах два и более потока могут выполняться одновременно.
2. **Для поддержки многопоточного программирования в Java предусмотрены класс ____________ и интерфейс ____________.**
   Для поддержки многопоточного программирования в Java предусмотрены класс `Thread` и интерфейс `Runnable`.
3. **В каких случаях при создании выполняемого объекта следует отдавать предпочтение расширению класса `Thread`, а не реализации интерфейса `Runnable`?**

Подклассы, производные от класса `Thread`, целесообразно создавать в тех случаях, когда помимо метода `run()` требуется переопределить другие методы данного класса.

4. Покажите, как с помощью метода `join()` можно организовать ожидание завершения потокового объекта `MyThrd`.

```
MyThrd.join()
```

5. Покажите, как установить приоритет потока `MyThrd` на три уровня выше нормального приоритета.

```
MyThrd.setPriority(Thread.NORM_PRIORITY+3);
```

6. Что произойдет, если в объявлении метода указать ключевое слово `synchronized`?

Если указать ключевое слово `synchronized` в объявлении метода, то в каждый момент времени этот метод будет вызываться только в одном потоке для любого заданного объекта его класса.

7. Методы `wait()` и `notify()` предназначены для обеспечения ______ ____________________.

взаимодействия потоков

8. Внесите в класс `TickTock` изменения для организации фактического отсчета времени. Первую половину секунды должен занимать вывод на экран слова "`Tick`", а вторую — вывод слова "`Tock`". Таким образом, сообщение "`Tick-Tock`" должно соответствовать одной секунде отсчитываемого времени. (Время переключения контекстов можно не учитывать.)

Для организации отчета времени достаточно добавить в класс `TickTock` вызовы метода `sleep()`, как показано ниже.

```
// Вариант класса TickTock, в который добавлены вызовы
// метода sleep() для организации отсчета времени.

class TickTock {

   String state; // содержит сведения о состоянии часов

   synchronized void tick(boolean running) {
      if(!running) { // остановить часы
         state = "ticked";
         notify(); // уведомить ожидающие потоки
         return;
      }

      System.out.print("Tick ");

      // Ожидать полсекунды
      try {
         Thread.sleep(500);
      } catch(InterruptedException exc) {
         System.out.println("Выполнение потока прервано.");
      }
```

```
        state = "ticked"; // установить текущее состояние
                          // после такта "тик"

        notify(); // разрешить выполнение метода tock()
        try {
           while(!state.equals("tocked"))
              wait(); // ожидать завершения метода tock()
        }
        catch(InterruptedException exc) {
           System.out.println("Выполнение потока прервано.");
        }
     }

     synchronized void tock(boolean running) {
        if(!running) { // остановить часы
           state = "tocked";
           notify(); // уведомить ожидающие потоки
           return;
        }

        System.out.println("Tock");

        // Ожидать полсекунды
        try {
           Thread.sleep(500);
        } catch(InterruptedException exc) {
           System.out.println("Выполнение потока прервано.");
        }

        state = "tocked"; // установить текущее состояние
                          // после такта "так"

        notify(); // разрешить выполнение метода tick()
        try {
           while(!state.equals("ticked"))
              wait(); // ожидать завершения метода tick()
        }
        catch(InterruptedException exc) {
           System.out.println("Выполнение потока прервано.");
        }
     }
  }
```

9. **Почему в новых программах на Java не следует применять методы `suspend()`, `resume()` и `stop()`?**
   Методы `suspend()`, `resume()` и `stop()` не рекомендуется применять, поскольку они могут стать причиной серьезных осложнений при выполнении программы.
10. **С помощью какого метода из класса `Thread` можно получить имя потока?**
    С помощью метода `getName()`.
11. **Какое значение возвращает метод `isAlive()`?**
    Метод возвращает значение `true`, если вызывающий поток выполняется, или значение `false`, если выполнение потока завершено.

Глава 12

1. Константы перечислимого типа иногда называют *самотипизированными*. Что это означает?
   Часть “само” в слове “самотипизированный” означает тип перечисления, в котором определена константа. Следовательно, константа перечислимого типа является объектом того перечисления, в которое она входит.
2. Какой класс автоматически наследуют перечисления?
   Все перечисления наследуют класс `Enum`.
3. Напишите для приведенного ниже перечисления программу, в которой метод `values()` используется для отображения списка констант и их значений.

```
enum Tools {
   SCREWDRIVER, WRENCH, HAMMER, PLIERS
}
```

Это задание имеет следующее решение.

```
enum Tools {
   SCREWDRIVER, WRENCH, HAMMER, PLIERS
}

class ShowEnum {
   public static void main(String args[]) {
      for(Tools d : Tools.values())
         System.out.print(d + " имеет порядковое значение " +
                          d.ordinal() + '\n');
   }
}
```

4. Созданную в упражнении 12.1 программу, имитирующую автоматизированный светофор, можно усовершенствовать, внеся ряд простых изменений, позволяющих выгодно воспользоваться возможностями перечислений. В исходной версии этой программы продолжительность отображения каждого цвета светофора регулировалась в классе `TrafficLightSimulator`, причем значения задержек были жестко запрограммированы в методе `run()`. Измените исходный код программы таким образом, чтобы продолжительность отображения каждого цвета светофора задавалась константами перечислимого типа `TrafficLightColor`. Для этого вам понадобятся конструктор, переменная экземпляра, объявленная как `private`, и метод `getDelay()`. Подумайте о том, как еще можно улучшить данную программу. (Подсказка: попробуйте отказаться от инструкции `switch` и воспользоваться порядковыми значениями каждого цвета для переключения светофора.)
   Усовершенствованная версия программы, имитирующей работу светофора, приведена ниже. В нее внесены два существенных изменения. Во-первых, величина задержки переключения связана теперь со значением перечислимого типа, что улучшает структуру кода. А во-вторых, в методе `run()` удалось обойтись без инструкции `switch`. Вместо этого методу `sleep()`

теперь передается вызов `tlc.getDelay()`, благодаря чему автоматически устанавливается задержка, соответствующая текущему цвету светофора.

```
// Усовершенствованная версия программы, имитирующей
// работу светофора. Значения задержки теперь хранятся
// в классе TrafficLightColor.

// Перечисление, представляющее цвета светофора
enum TrafficLightColor {
   RED(12000), GREEN(10000), YELLOW(2000);

   private int delay;

   TrafficLightColor(int d) {
      delay = d;
   }

   int getDelay() { return delay; }
}

// Имитация автоматизированного светофора
class TrafficLightSimulator implements Runnable {
   private TrafficLightColor tlc; // текущий цвет светофора
   boolean stop = false; // для остановки имитации установить в true
   boolean changed = false; // true, если светофор переключился

   TrafficLightSimulator(TrafficLightColor init) {
      tlc = init;
   }

   TrafficLightSimulator() {
      tlc = TrafficLightColor.RED;
   }

   // Запуск имитации автоматизированного светофора
   public void run() {
      while(!stop) {
         // По сравнению с предыдущей версией программы
         // код значительно упростился!
         try {
             Thread.sleep(tlc.getDelay());
         } catch(InterruptedException exc) {
             System.out.println(exc);
         }

         changeColor();
      }
   }

   // Переключение цвета светофора
   synchronized void changeColor() {
      switch(tlc) {
         case RED:
            tlc = TrafficLightColor.GREEN;
            break;
```

```
            case YELLOW:
               tlc = TrafficLightColor.RED;
               break;
            case GREEN:
               tlc = TrafficLightColor.YELLOW;
         }

         changed = true;
         notify(); // уведомить о переключении цвета светофора
      }

      // Ожидание переключения цвета светофора
      synchronized void waitForChange() {
         try {
            while(!changed)
               wait(); // ожидать переключения цвета светофора
            changed = false;
         } catch(InterruptedException exc) {
            System.out.println(exc);
         }
      }

      // Возврат текущего цвета
      synchronized TrafficLightColor getColor() {
         return tlc;
      }

      // Прекращение имитации светофора
      synchronized void cancel() {
         stop = true;
      }
   }

   class TrafficLightDemo {
      public static void main(String args[]) {
         TrafficLightSimulator tl =
            new TrafficLightSimulator(TrafficLightColor.GREEN);

         Thread thrd = new Thread(tl);
         thrd.start();
         for(int i=0; i < 9; i++) {
            System.out.println(tl.getColor());
            tl.waitForChange();
         }

         tl.cancel();
      }
   }
```

5. **Что такое упаковка и распаковка? В каких случаях выполняется автоупаковка и автораспаковка?**

   Упаковка означает сохранение значения простого типа в объекте оболочки, а распаковка — извлечение значения из объекта оболочки. Автоупаковка означает автоматическую упаковку значения без явного создания объекта,

тогда как при автораспаковке значение простого типа автоматически извлекается из объекта оболочки без явного вызова соответствующего метода, например `intValue()`.

6. Измените следующий фрагмент кода таким образом, чтобы в нем выполнялась автоупаковка:

```
Double val = Double.valueOf(123.0);
```

Это задание имеет следующее решение:

```
Double val = 123.0;
```

7. Объясните, что такое статический импорт.

   Статический импорт означает размещение статических членов класса или интерфейса в глобальном пространстве имен. Это позволяет использовать статические члены без указания имени соответствующего класса или интерфейса.

8. Какие действия выполняет приведенная ниже инструкция?

```
import static java.lang.Integer.parseInt;
```

Эта инструкция помещает в глобальное пространство имен метод `parseInt()` класса оболочки типа `Integer`.

9. Следует ли использовать статический импорт применительно к конкретным ситуациям или желательно импортировать статические члены всех классов?

   Статический импорт уместен только в отдельных случаях. Если доступным окажется слишком много статических членов, это может привести к конфликтам имен и нарушению структуры кода.

10. Синтаксис аннотации основывается на ______________.

    интерфейсе

11. Какая аннотация называется маркерной?

    Маркерной называют аннотацию, не имеющую аргументов.

12. Справедливо ли следующее утверждение: “Аннотации применимы только к методам”?

    Нет. Аннотировать можно любое объявление.

Глава 13

1. Обобщения очень важны, поскольку позволяют создавать код, который:

   1) обеспечивает безопасность типов;

   2) пригоден для повторного использования;

   3) отличается высокой надежностью;

   4) обладает всеми перечисленными выше свойствами.

   Ответ: г) код обладает всеми перечисленными выше свойствами.

2. Можно ли указывать простой тип в качестве аргумента типа?

   Нет, нельзя. В качестве аргументов типа можно указывать только типы объектов.

3. Как объявить класс `FlightSched` с двумя параметрами типа?

   Это задание имеет следующее решение:

```
class FlightSched<T, V> {
```

4. Измените ваш ответ на вопрос 3 таким образом, чтобы второй параметр типа обозначал подкласс, производный от класса `Thread`.

   Это задание имеет следующее решение:

```
class FlightSched<T, V extends Thread> {
```

5. Внесите изменения в класс `FlightSched` таким образом, чтобы второй параметр типа стал подклассом первого параметра типа.

   Это задание имеет следующее решение:

```
class FlightSched<T, V extends T> {
```

6. Что обозначает знак `?` в обобщениях?

   Знак `?` обозначает метасимвольный аргумент, который соответствует любому допустимому типу.

7. Может ли шаблон аргумента быть ограниченным?

   Да. Шаблон аргумента может ограничиваться как сверху, так и снизу.

8. У обобщенного метода `MyGen()` имеется один параметр типа, определяющий тип передаваемого ему аргумента. Этот метод возвращает также объект, тип которого соответствует параметру типа. Как должен быть объявлен метод `MyGen()`?

   Это задание имеет следующее решение:

```
<T> T MyGen(T o) { // ...
```

9. Допустим, обобщенный интерфейс объявлен так:

```
interface IGenIF<T, V extends T> { // ...
```

   Напишите объявление класса `MyClass`, который реализует интерфейс `IGenIF`.

   Это задание имеет следующее решение:

```
class MyClass<T, V extends T> implements IGenIF<T, V> { // ...
```

10. Допустим, имеется обобщенный класс `Counter<T>`. Как создать объект его базового типа?

    Для того чтобы получить базовый тип из обобщенного класса `Counter<T>`, достаточно указать его имя, не обозначая тип:

```
Counter x = new Counter;
```

11. Существуют ли параметры типа на стадии выполнения программы?

    Нет. Все параметры типа удаляются на стадии компиляции и заменяются соответствующими приводимыми типами. Этот процесс называется очисткой.

12. Видоизмените ответ на вопрос 10 в упражнении для самопроверки из главы 9 таким образом, чтобы сделать класс обобщенным. Для этого создайте интерфейс стека `IGenStack`, объявив в нем обобщенные методы `push()` и `pop()`.

```
// Обобщенный стек

interface IGenStack<T> {
   void push(T obj) throws StackFullException;
   T pop() throws StackEmptyException;
}

// Исключение, возникающее при переполнении стека
class StackFullException extends Exception {
   int size;

   StackFullException(int s) { size = s; }

   public String toString() {
      return "\nСтек заполнен. Максимальный размер стека: " + size;
   }
}

// Исключение, возникающее при обращении к пустому стеку
class StackEmptyException extends Exception {

   public String toString() {
      return "\nСтек пуст.";
   }
}

// Класс, реализующий стек для хранения символов
class GenStack<T> implements IGenStack<T> {
   private T stck[]; // массив для хранения элементов стека
   private int tos;  // вершина стека

   // Создать пустой стек заданного размера
   GenStack(T[] stckArray) {
      stck = stckArray;
      tos = 0;
   }

   // Создать один стек на основе другого стека
   GenStack(T[] stckArray, GenStack<T> ob) {
      tos = ob.tos;
      stck = stckArray;

      try {
         if(stck.length < ob.stck.length)
            throw new StackFullException(stck.length);
      }
      catch(StackFullException exc) {
         System.out.println(exc);
      }

      // Скопировать элементы
      for(int i=0; i < tos; i++)
```

```
            stck[i] = ob.stck[i];
    }

    // Создать стек с начальными значениями
    GenStack(T[] stckArray, T[] a) {
        stck = stckArray;

        for(int i = 0; i < a.length; i++) {
            try {
                push(a[i]);
            }
            catch(StackFullException exc) {
                System.out.println(exc);
            }
        }
    }

    // Поместить объект в стек
    public void push(T obj) throws StackFullException {
        if(tos==stck.length)
            throw new StackFullException(stck.length);

        stck[tos] = obj;
        tos++;
    }

    // Извлечь объект из стека
    public T pop() throws StackEmptyException {
        if(tos==0)
            throw new StackEmptyException();

        tos--;
        return stck[tos];
    }
}

// Демонстрация использования класса GenStack
class GenStackDemo {
    public static void main(String args[]) {
        // Создать пустой стек на 10 элементов типа Integer
        Integer iStore[] = new Integer[10];
        GenStack<Integer> stk1 = new GenStack<Integer>(iStore);

        // Создать стек из массива
        String name[] = {"Один", "Два", "Три"};
        String strStore[] = new String[3];
        GenStack<String> stk2 = new GenStack<String>(strStore, name);

        String str;
        int n;
```

```
        try {
            // Занести ряд значений в стек stk1
            for(int i=0; i < 10; i++)
                stk1.push(i);
        } catch(StackFullException exc) {
            System.out.println(exc);
        }

        // Создать один стек на основе другого стека
        String strStore2[] = new String[3];
        GenStack<String> stk3 = new GenStack<String>(strStore2, stk2);

        try {
            // Отобразить стеки
            System.out.print("Содержимое stk1: ");
            for(int i=0; i < 10; i++) {
                n = stk1.pop();
                System.out.print(n + " ");
            }

            System.out.println("\n");

            System.out.print("Содержимое stk2: ");
            for(int i=0; i < 3; i++) {
                str = stk2.pop();
                System.out.print(str + " ");
            }

            System.out.println("\n");

            System.out.print("Содержимое stk3: ");
            for(int i=0; i < 3; i++) {
                str = stk3.pop();
                System.out.print(str + " ");
            }

        } catch(StackEmptyException exc) {
            System.out.println(exc);
        }

        System.out.println();
    }
}
```

13. Что означает пара угловых скобок (<>)?

Угловые скобки обозначают пустой список аргументов типа.

14. Как упростить приведенную ниже строку кода?

```
MyClass<Double,String> obj = new MyClass<Double,String>(1.1,"Привет");
```

Эту строку кода можно упростить, используя ромбовидный оператор следующим образом:

```
MyClass<Double,String> obj = new MyClass<>(1.1,"Привет");
```

Глава 14

1. Что такое лямбда-оператор?
   Это оператор ->.
2. Что такое функциональный интерфейс?
   Это интерфейс, который имеет один и только один абстрактный метод.
3. Какая связь существует между функциональными интерфейсами и лямбда-выражениями?
   Лямбда-выражение предоставляет реализацию абстрактного метода, определяемого функциональным интерфейсом. Функциональный интерфейс определяет целевой тип.
4. Назовите два общих типа лямбда-выражений.
   Лямбда-выражения бывают строчными и блочными. Строчное лямбда-выражение определяет одиночное выражение, значение которого возвращается лямбда-оператором. Блочное лямбда-выражение содержит блок кода. Его значение определяется инструкцией `return`.
5. Составьте лямбда-выражение, которое возвращает значение `true`, если число принадлежит к диапазону 10–20, включая граничные значения.

```
(n) -> (n > 9 && n < 21)
```

6. Создайте функциональный интерфейс, способный поддерживать лямбда-выражение, предложенное в п. 5. Назовите интерфейс `MyTest`, а его абстрактный метод — `testing()`.

```
interface MyTest {
   boolean testing(int n);
}
```

7. Создайте блочное лямбда-выражение для вычисления факториала целого числа. Продемонстрируйте его использование. В качестве функционального интерфейса используйте интерфейс `NumericFunc`, который рассматривался в этой главе.

```
interface NumericFunc {
   int func(int n);
}

class FactorialLambdaDemo {
   public static void main(String args[])
   {

      // Это блочное выражение вычисляет факториал
      // целочисленного значения
      NumericFunc factorial = (n) -> {
         int result = 1;

         for(int i=1; i <= n; i++)
            result = i * result;
```

```
            return result;
        };

        System.out.println("Факториал 3 равен " + factorial.func(3));
        System.out.println("Факториал 5 равен " + factorial.func(5));
        System.out.println("Факториал 9 равен " + factorial.func(9));
    }
}
```

8. Создайте обобщенный функциональный интерфейс `MyFunc<T>`. Назовите его абстрактный метод `func()`. Метод `func()` должен иметь параметр типа `T` и возвращать ссылку типа `T`. (Таким образом, интерфейс `MyFunc` должен представлять собой обобщенную версию интерфейса `NumericFunc`, который рассматривался в этой главе.) Продемонстрируйте его использование, переработав свое решение для п. 7 таким образом, чтобы вместо интерфейса `NumericFunc` в нем использовался интерфейс `MyFunc<T>`.

```
interface MyFunc<T> {
    T func(T n);
}

class FactorialLambdaDemo {
    public static void main(String args[])
    {

        // Это блочное лямбда-выражение вычисляет факториал
        // целочисленного значения
        MyFunc<Integer> factorial = (n) -> {
            int result = 1;

            for(int i=1; i <= n; i++)
                result = i * result;

            return result;
        };

        System.out.println("Факториал 3 равен " + factorial.func(3));
        System.out.println("Факториал 5 равен " + factorial.func(5));
        System.out.println("Факториал 9 равен " + factorial.func(9));
    }
}
```

9. Используя программу, созданную в упражнении 14.1, создайте лямбда-выражение, которое удаляет все пробелы из заданной строки и возвращает результат. Продемонстрируйте работу лямбда-выражения, передав его методу `changeStr()`.

Вот лямбда-выражение, удаляющее пробелы. Оно используется для инициализации ссылочной переменной `remove`.

```
StringFunc remove = (str) -> {
    String result = "";
```

```
   for(int i = 0; i < str.length(); i++)
      if(str.charAt(i) != ' ') result += str.charAt(i);

   return result;
};
```

Вот пример его использования:

```
outStr = changeStr(remove, inStr);
```

10. Можно ли использовать в лямбда-выражении локальную переменную? Если да, то какие при этом существуют ограничения?

Можно, но переменная должна быть объявлена как `final`.

11. Справедливо ли следующее утверждение: "Если лямбда-выражение может генерировать проверяемое исключение, то абстрактный метод функционального интерфейса должен содержать спецификацию `throws`, в которой указано данное исключение"?

Справедливо.

12. Что такое ссылка на метод?

Ссылка на метод — это способ обращения к методу без его вызова.

13. При вычислении ссылки на метод создается экземпляр ____________ ________________, предоставляемого целевым контекстом.

функционального интерфейса

14. Предположим, имеется класс `MyClass`, содержащий статический метод `myStaticMethod()`. Продемонстрируйте, как можно указать ссылку на метод `myStaticMethod()`.

```
myClass::myStaticMethod
```

15. Предположим, имеется класс `MyClass`, содержащий объектный метод `myInstMethod()`, и относящийся к этому классу объект `mcObj`. Продемонстрируйте, как можно создать ссылку на метод `myInstMethod()`, ассоциированный с объектом `mcObj`.

```
mcObj::myInstMethod
```

16. В программе `MethodRefDemo2` добавьте в класс `MyIntNum` новый метод `hasCommonFactor()`. Этот метод должен возвращать `true`, если его аргумент типа `int` и значение, которое хранится в вызывающем объекте `MyIntNum`, имеют по крайней мере один общий делитель. Продемонстрируйте работу метода `hasCommonFactor()`, используя ссылку на метод.

Ниже приведено объявление класса `MyIntNum`, в который добавлен метод `hasCommonFactor()`.

```
class MyIntNum {
   private int v;

   MyIntNum(int x) { v = x; }

   int getNum() { return v; }
```

```
    // Вернуть true, если n является делителем v
    boolean isFactor(int n) {
       return (v % n) == 0;
    }

    boolean hasCommonFactor(int n) {
       for(int i=2; i < v/i; i++)
          if( ((v % i) == 0) && ((n % i) == 0) ) return true;

       return false;
    }
}
```

Ниже приведен пример использования этого класса посредством ссылки на метод.

```
ip = myNum::hasCommonFactor;
result = ip.test(9);
if(result) System.out.println("Общий делитель найден.");
```

17. Как определяется ссылка на конструктор?

Ссылка на конструктор создается путем указания имени класса, после которого вслед за символами `::` указывается оператор `new`. Например: `MyClass::new`.

18. В каком пакете Java содержатся определения встроенных функциональных интерфейсов?

`java.util.function`

Глава 15

1. В широком смысле модули помогают определить зависимость одной единицы кода от другой. Верно ли это утверждение?

Да.

2. Какое ключевое слово применяется для объявления модуля?

`module`

3. Ключевые слова, поддерживающие модули, являются контекстно-зависимыми. Объясните, что это значит.

Контекстно-зависимое ключевое слово распознается как ключевое слово только в определенных ситуациях, связанных с его использованием, и больше нигде. Поскольку речь идет о ключевых словах, поддерживающих модуль, они распознаются как ключевые слова только в объявлении модуля.

4. Что представляет собой файл `module-info.java` и в чем его важность?

Файл `module-info.java` содержит объявление модуля.

5. Какое ключевое слово используется для объявления зависимости одного модуля от другого?

`requires`

6. Чтобы открытые компоненты пакета стали доступны за пределами модуля, в котором они содержатся, пакет следует указать в инструкции ___________.
   `exports`
7. Почему путь к модулю важно указывать при компиляции или выполнении модульного приложения?
   Путь к модулю определяет местонахождение модулей приложения.
8. Что делает инструкция `requires transitive`?
   С помощью инструкции `requires transitive` создается неявная (транзитивная) зависимость, когда любой модуль, зависящий от текущего, также зависит от того модуля, который указан в инструкции `requires transitive`.
9. Инструкция `exports` экспортирует модуль или пакет?
   Инструкция `exports` экспортирует пакет.
10. Какая ошибка может возникнуть, если в первом примере модуля удалить строку

    ```
    exports appfuncs.simplefuncs;
    ```

    из файла `module-info` для модуля `appfuncs`, а затем попытаться скомпилировать программу?
    Компилятор сообщает о том, что пакета `SimpleMathFuncs` не существует. Поскольку этот пакет требуется для приложения `MyModAppDemo`, оно не будет скомпилировано.
11. Какие ключевые слова применяются для работы с модульными службами?
    `provides`, `uses` и `with`.
12. Служба определяет общую функциональность программной единицы с помощью интерфейса или абстрактного класса. Верно ли это утверждение?
    Да.
13. Провайдер службы _____________ службу.
    реализует
14. Какой класс используется для загрузки службы?
    `ServiceLoader`
15. Может ли модульная зависимость быть необязательной на этапе выполнения? Если да, то каким образом?
    Да, с помощью инструкции `exports static`.
16. Вкратце объясните, для чего используются ключевые слова `open` и `opens`.
    Добавление ключевого слова `open` в объявление модуля разрешает доступ к его пакетам на этапе выполнения, в том числе путем рефлексии, независимо от того, были ли пакеты экспортированы. Инструкция `opens` обеспечивает доступ к пакету на этапе выполнения, в том числе для целей рефлексии.

Глава 16

1. Компоненты AWT являются тяжеловесными, а компоненты Swing — ________________.
   легковесными
2. Можно ли изменить стиль оформления компонента Swing? Если да, то какое средство позволяет это сделать?
   Да, можно. Это позволяют сделать подключаемые стили оформления Swing.
3. Какой контейнер верхнего уровня чаще всего используется в приложениях?
   Контейнер `JFrame`.
4. Контейнер верхнего уровня содержит несколько панелей. На какой из них располагаются компоненты?
   На панели содержимого.
5. Как создать метку, отображающую сообщение `"Выберите элемент списка"`?
   `JLabel("Выберите элемент списка")`
6. В каком потоке должно осуществляться любое взаимодействие с компонентами графического пользовательского интерфейса?
   В потоке диспетчеризации событий.
7. Какая команда действия связывается по умолчанию с компонентом `JButton`? Как изменить команду действия?
   По умолчанию строка команды действия содержит текст надписи на кнопке. Команду действия можно изменить, вызвав метод `setActionCommand()`.
8. Какое событие генерируется при щелчке на кнопке?
   Событие `ActionEvent`.
9. Как создать текстовое поле шириной 32 столбца?
   `JTextField(32)`
10. Можно ли задать команду действия для компонента `JTextField`? Если да, то как это сделать?
    Да, можно. Для этого достаточно вызвать метод `setActionCommand()`.
11. С помощью какого компонента Swing можно создать флажок? Какое событие генерируется при установке или сбросе флажка?
    Флажок создается с помощью компонента `JCheckBox`. При установке или сбросе флажка генерируется событие `ItemEvent`.
12. Компонент `JList` отображает список элементов, которые может выбирать пользователь. Верно или неверно?
    Верно.
13. Какое событие генерируется при выборе или отмене выбора элемента из списка типа `JList`?
    Событие `ListSelectionEvent`.

14. В каком методе задается режим выбора элементов списка `JList`? С помощью какого метода можно получить индекс первого выбранного элемента?
Режим выбора элементов списка задается в методе `setSelectionMode()`. Метод `getSelectedIndex()` возвращает индекс первого выбранного элемента.

15. Добавьте в утилиту сравнения файлов, созданную в упражнении 16.1, флажок со следующей пояснительной надписью: `Показывать позицию расхождения`. Если этот флажок установлен, программа должна отображать позицию, в которой обнаружено первое расхождение в содержимом сравниваемых файлов.

```
/*
   Упражнение 16.1.

   Утилита сравнения файлов на основе Swing.

   В этой версии предусмотрен флажок, установка которого задает
   отображение позиции первого расхождения в содержимом
   сравниваемых файлов.
*/

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

class SwingFC implements ActionListener {

   JTextField jtfFirst; // хранит имя первого файла
   JTextField jtfSecond; // хранит имя второго файла

   JButton jbtnComp; // кнопка для запуска операции сравнения файлов

   JLabel jlabFirst, jlabSecond; // метки, отображающие
                                 // подсказки для пользователя
   JLabel jlabResult; // метка для отображения результата
                      // сравнения и сообщений об ошибках

   JCheckBox jcbLoc; // установить для отображения позиции
                     // первого несовпадения файлов

   SwingFC() {

      // Создать новый контейнер JFrame
      JFrame jfrm = new JFrame("Сравнить файлы");

      // Задать объект FlowLayout для менеджера компоновки
      jfrm.setLayout(new FlowLayout());

      // Задать исходные размеры фрейма
      jfrm.setSize(200, 190);
```

```
        // Прекратить работу программы, если
        // пользователь закрывает приложение
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создать поля для ввода имен файлов
        jtfFirst = new JTextField(14);
        jtfSecond = new JTextField(14);

        // Задать команды действия для текстовых полей
        jtfFirst.setActionCommand("ФайлА");
        jtfSecond.setActionCommand("ФайлБ");

        // Создать кнопку сравнения
        JButton jbtnComp = new JButton("Сравнить");

        // Добавить слушатель событий для кнопки
        jbtnComp.addActionListener(this);

        // Создать метки
        jlabFirst = new JLabel("Первый файл ");
        jlabSecond = new JLabel("Второй файл: ");
        jlabResult = new JLabel("");

        // Создать флажок
        jcbLoc = new JCheckBox("Показать позицию расхождения");

        // Добавить компоненты на панель содержимого
        jfrm.add(jlabFirst);
        jfrm.add(jtfFirst);
        jfrm.add(jlabSecond);
        jfrm.add(jtfSecond);
        jfrm.add(jcbLoc);
        jfrm.add(jbtnComp);
        jfrm.add(jlabResult);

        // Отобразить фрейм
        jfrm.setVisible(true);
    }

    // Сравнить файлы после щелчка на кнопке
    public void actionPerformed(ActionEvent ae) {
        int i=0, j=0;
        int count = 0;

        // Сначала убедиться в том, что введены имена обоих файлов
        if(jtfFirst.getText().equals("")) {
            jlabResult.setText("Отсутствует имя первого файла.");
            return;
        }

        if(jtfSecond.getText().equals("")) {
            jlabResult.setText("Отсутствует имя второго файла.");
```

```
            return;
        }

        // Сравнить файлы, используя инструкцию try с ресурсами
        try (FileInputStream f1 = new
                                FileInputStream(jtfFirst.getText());
             FileInputStream f2 = new
                                FileInputStream(jtfSecond.getText()))
        {
            // Сравнить содержимое обоих файлов
            do {
                i = f1.read();
                j = f2.read();
                if(i != j) break;
                count++;
            } while(i != -1 && j != -1);

            if(i != j) (
                if(jcbLoc.isSelected())
                    jlabResult.setText("Файлы различаются, начиная
                                         с позиции " + count);
                else
                    jlabResult.setText("Файлы отличаются.");
            }
            else
                jlabResult.setText("Сравниваемые файлы совпадают.");

        } catch(IOException exc) {
            jlabResult.setText("Ошибка файла ");
        }
    }

    public static void main(String args[]) {
        // Создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SwingFC();
            }
        });
    }
}
```

16. Измените программу `ListDemo` таким образом, чтобы она допускала выбор нескольких элементов списка.

```
// Демонстрация выбора нескольких элементов из списка
// с помощью компонента Jlist

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

```
class ListDemo implements ListSelectionListener {

   JList<String> jlst;
   JLabel jlab;
   JScrollPane jscrlp;

   // Создать массив имен
   String names[] = { "Мария", "Иван", "Светлана",
                      "Александр", "Евгения", "Наталья",
                      "Аркадий", "Валентина", "Борис",
                      "Андрей", "Степан", "Владислав" };

   ListDemo() {
      // Создать новый контейнер JFrame
      JFrame jfrm = new JFrame("Демонстрация списка");

      // Задать объект FlowLayout для менеджера компоновки
      jfrm.setLayout(new FlowLayout());

      // Задать исходные размеры фрейма
      jfrm.setSize(200, 160);

      // Прекратить работу программы, если
      // пользователь закрывает приложение
      jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Создать компонент JList
      jlst = new JList<String>(names);

      // Удаление следующей строки кода задаст режим группового
      // выбора элементов из списка (этот режим
      // устанавливается для компонента JList по умолчанию)
//    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

      // Добавить список на панель с полосами прокрутки
      jscrlp = new JScrollPane(jlst);

      // Задать предпочтительные размеры прокручиваемой панели
      jscrlp.setPreferredSize(new Dimension(120, 90));

      // Создать метку для отображения результатов выбора
      jlab = new JLabel("Выберите имя");

      // Добавить обработчик для событий списка
      jlst.addListSelectionListener(this);

      // Добавить список и метку на панель содержимого
      jfrm.add(jscrlp);
      jfrm.add(jlab);

      // Отобразить фрейм
      jfrm.setVisible(true);
   }
```

```
    // Обработка событий списка
    public void valueChanged(ListSelectionEvent le) {
       // Получить индексы тех элементов, выбор которых был сделан
       // или отменен в списке
       int indices[] = jlst.getSelectedIndices();

       // Отобразить результат выбора, если был выбран один
       // или несколько элементов из списка
       if(indices.length != 0) {
          String who = "";

          // Создать строку из выбранных имен
          for(int i : indices)
             who += names[i] + " ";

          jlab.setText("Текущие выделения: " + who);
       }
       else // иначе еще раз предложить сделать выбор
          jlab.setText("Выберите имя");
    }

    public static void main(String args[]) {
       // Создать фрейм в потоке диспетчеризации событий
       SwingUtilities.invokeLater(new Runnable() {
          public void run() {
             new ListDemo();
          }
       });
    }
 }
```

Глава 17

1. **Назовите имя пакета верхнего уровня библиотеки JavaFX.**
   `javafx`
2. **Двумя центральными понятиями в JavaFX являются платформа и сцена. Какие классы их инкапсулируют?**
   `Stage` и `Scene`.
3. **Граф сцены состоит из ________________________________.**
   узлов
4. **Базовым классом для всех узлов служит класс ______________.**
   `Node`
5. **Какой класс должны расширять все приложения JavaFX?**
   `Application`
6. **Какие три метода управляют жизненным циклом приложения JavaFX?**
   `init()`, `start()` и `stop()`.

7. **В каком из методов, управляющих жизненным циклом, возможно создание платформы приложения?**
   `start()`
8. **Метод `launch()` вызывается для запуска автономного приложения JavaFX. Верно или неверно?**
   Верно.
9. **Назовите классы JavaFX, которые реализуют метку и кнопку.**
   `Label` и `Button`.
10. **Одним из способов прекращения работы автономного приложения JavaFX является вызов метода `Platform.exit()`. Класс `Platform` находится в пакете `javafx.Application`. При вызове метода `exit()` работа программы немедленно прекращается. Учитывая это, измените программу `JavaFXEventDemo`, представленную в данной главе, таким образом, чтобы она отображала две кнопки: Выполнить и Выход. При нажатии кнопки Выполнить программа должна вывести соответствующее сообщение в метке. При нажатии кнопки Выход приложение должно завершить свою работу. В обработчиках событий используйте лямбда-выражения.**

```
// Демонстрация использования метода Platform.exit()

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

   Label response;

   public static void main(String[] args) {

      // Запустить приложение JavaFX, вызвав метод launch()
      launch(args);
   }

   // Переопределить метод start()
   public void start(Stage myStage) {

      // Задать заголовок окна приложения
      myStage.setTitle("Использование метода Platform.exit().");

      // Использовать компоновку FlowPane для корневого узла.
      // В данном случае величина вертикального и горизонтального
      // зазоров составляет 10.
      FlowPane rootNode = new FlowPane(10, 10);
```

```
        // Центрировать компоненты на сцене
        rootNode.setAlignment(Pos.CENTER);

        // Создать сцену
        Scene myScene = new Scene(rootNode, 300, 100);

        // Установить сцену на платформе
        myStage.setScene(myScene);

        // Создать метку
        response = new Label("Нажмите кнопку");

        // Создать две кнопки
        Button btnRun = new Button("Выполнить");
        Button btnExit = new Button("Выход");

        // Обработать события действий для кнопки "Выполнить"
        btnRun.setOnAction((ae) ->
                        response.setText("Вы нажали Выполнить."));

        // Обработать события действий для кнопки "Выход"
        btnExit.setOnAction((ae) -> Platform.exit());

        // Добавить метку и кнопки в граф сцены
        rootNode.getChildren().addAll(btnRun, btnExit, response);

        // Отобразить платформу вместе с ее сценой
        myStage.show();
    }
}
```

11. Какой компонент JavaFX реализует флажок?

`CheckBox`

12. Класс `ListView` — это компонент, который отображает список файлов, находящихся в некотором каталоге локальной файловой системы. Верно или неверно?

Неверно. Компонент `ListView` отображает список элементов, доступных для выбора пользователем.

13. Преобразуйте Swing-программу для сравнения файлов из упражнения 16.1 в приложение JavaFX. При этом воспользуйтесь предоставляемой в JavaFX возможностью запускать события действий для кнопки программными средствами. Это делается путем вызова метода `fire()` для экземпляра кнопки. К примеру, если имеется экземпляр класса `Button`, который вы назвали `myButton`, то для запуска события необходимо вызвать метод `myButton.fire()`. Воспользуйтесь этим при реализации обработчиков событий для текстовых полей, в которых хранятся имена сравниваемых файлов. В тех случаях, когда пользователь нажимает клавишу <Enter> и при этом фокус ввода находится в одном из указанных текстовых полей,

запускайте событие действия для кнопки **Сравнить**. После этого код обработчика событий для кнопки **Сравнить** должен выполнить сравнение файлов.

```
// Реализация средствами JavaFX версии утилиты сравнения файлов,
// представленной в упражнении 16.1.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import java.io.*;

public class JavaFXFileComp extends Application {

   TextField tfFirst; // хранит имя первого файла
   TextField tfSecond; // хранит имя второго файла

   Button btnComp; // кнопка для запуска операции сравнения файлов

   Label labFirst, labSecond; // метки, отображающие
                              // подсказки для пользователя
   Label labResult; // метка для отображения результата
                    // сравнения и сообщений об ошибках

   public static void main(String[] args) {

      // Запустить приложение JavaFX, вызвав метод launch()
      launch(args);
   }

   // Переопределить метод start()
   public void start(Stage myStage) {

      // Задать заголовок окна приложения
      myStage.setTitle("Сравнить файлы");

      // Использовать компоновку FlowPane для корневого узла.
      // В данном случае величина вертикального и горизонтального
      // зазоров составляет 10.
      FlowPane rootNode = new FlowPane(10, 10);

      // Центрировать компоненты на сцене
      rootNode.setAlignment(Pos.CENTER);

      // Создать сцену
      Scene myScene = new Scene(rootNode, 180, 180);
```

```
// Установить сцену на платформе
myStage.setScene(myScene);

// Создать текстовые поля для имен файлов
tfFirst = new TextField();
tfSecond = new TextField();

// Задать предпочтительные размеры столбцов
tfFirst.setPrefColumnCount(12);
tfSecond.setPrefColumnCount(12);

// Задать подсказки для имен файлов
tfFirst.setPromptText("Введите имя файла.");
tfSecond.setPromptText("Введите имя файла.");

// Создать кнопку сравнения
btnComp = new Button("Сравнить");

// Создать метки
labFirst = new Label("Первый файл: ");
labSecond = new Label("Второй файл: ");
labResult = new Label("");

// Использовать лямбда-выражения для обработки событий
// действий, связанных с текстовыми полями. Эти обработчики
// просто запускают событие для кнопки "Сравнить".
tfFirst.setOnAction( (ae) -> btnComp.fire());
tfSecond.setOnAction( (ae) -> btnComp.fire());

// Обработать события действий для кнопки "Сравнить"
btnComp.setOnAction(new EventHandler<ActionEvent>() {
   public void handle(ActionEvent ae) {
      int i=0, j=0;

      // Сначала убедиться в том, что введены
      // имена обоих файлов
      if(tfFirst.getText().equals("")) {
         labResult.setText("Отсутствует имя первого файла.");
         return;
      }
      if(tfSecond.getText().equals("")) {
         labResult.setText("Отсутствует имя второго файла.");
         return;
      }

      // Сравнить файлы, используя инструкцию try с ресурсами
      try (FileInputStream f1 = new
                        FileInputStream(tfFirst.getText());
           FileInputStream f2 = new
                        FileInputStream(tfSecond.getText()))
      {
```

```
                // Сравнить содержимое обоих файлов
                do {
                   i = f1.read();
                   j = f2.read();
                   if(i != j) break;
                } while(i != -1 && j != -1);

                if(i != j)
                   labResult.setText("Файлы отличаются.");
                else
                   labResult.setText("Файлы одинаковы.");

             } catch(IOException exc) {
                labResult.setText("Ошибка файла");
             }
          }
       });

       // Добавить компоненты в граф сцены
       rootNode.getChildren().addAll(labFirst, tfFirst, labSecond,
                                     tfSecond, btnComp, labResult);

       // Отобразить платформу вместе с ее сценой
       myStage.show();
    }
}
```

14. Модифицируйте программу `EffectsAndTransformsDemo` таким образом, чтобы размытие применялось также к кнопке **Повернуть**. Задайте для ширины и высоты области размытия значение 5, а для счетчика итераций — значение 2.

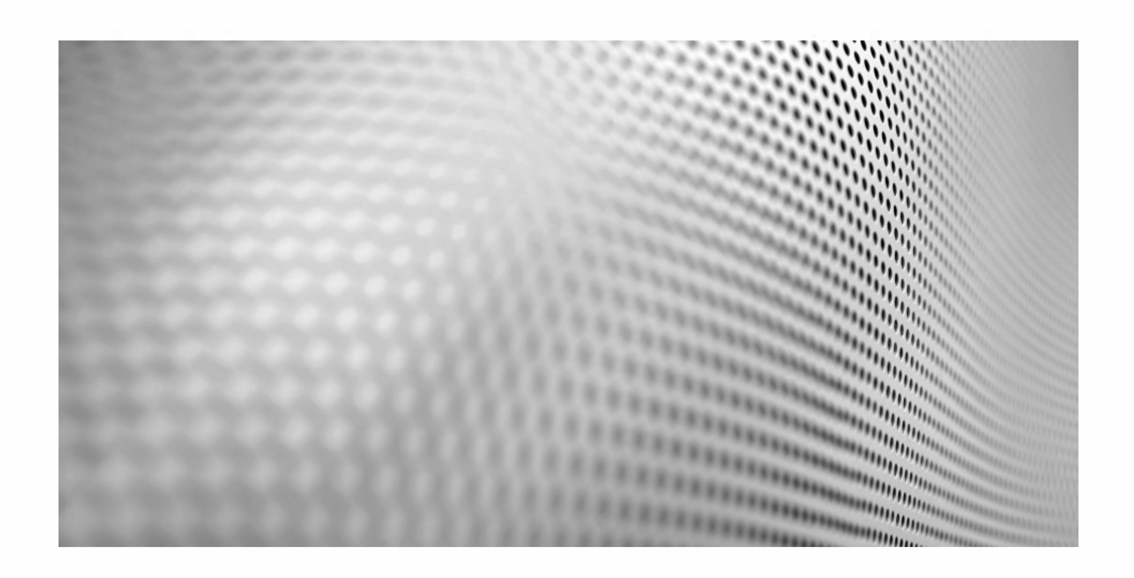
Чтобы добавить эффект размытия для кнопки **Повернуть**, прежде всего создайте экземпляр класса `BoxBlur`:

```
BoxBlur rotateBlur = new BoxBlur(5.0, 5.0, 2);
```

Затем добавьте следующую строку кода:

```
btnRotate.setEffect(rotateBlur);
```

После внесения указанных изменений изображение кнопки будет размываться, и ее можно будет поворачивать на заданный угол.



Приложение Б

Применение документирующих комментариев в Java

Как объяснялось в главе 1, в Java поддерживаются три вида комментариев. Первым двум соответствуют символы // и /* */, а третий вид называется *документирующими комментариями*. Такие комментарии начинаются символами /** и заканчиваются символами */. Документирующие комментарии позволяют включать сведения о программе в исходный код самой программы. Для извлечения этих сведений и их последующего преобразования в формат HTML-документа служит утилита javadoc, входящая в состав JDK. Документирующие комментарии — удобный способ документирования прикладных программ. Вам, вероятно, уже встречалась документация, сформированная утилитой javadoc, поскольку именно такой способ применяется для составления документации на библиотеку Java API. Начиная с JDK 9 утилита javadoc включает поддержку модулей.

Дескрипторы javadoc

Утилита javadoc распознает и обрабатывает в документирующих комментариях следующие дескрипторы.

Дескриптор	Описание
@author	Указывает автора программы
{@code}	Отображает данные шрифтом, предназначенным для вывода исходного кода, не выполняя преобразований в формат HTML-документа
@deprecated	Указывает на то, что элемент программы не рекомендован к применению
{@docRoot}	Указывает путь к корневому каталогу документации
@exception	Обозначает исключение, генерируемое методом
@hidden	Предотвращает появление элемента в документации (добавлено в JDK 9)
{@index}	Определяет термин для индексирования (добавлено в JDK 9)
{@inheritDoc}	Наследует комментарии от ближайшего суперкласса
{@link}	Вставляет встроенную ссылку на другую тему
{@linkplain}	Вставляет встроенную ссылку на другую тему, но ссылка отображается тем же шрифтом, что и простой текст
{@literal}	Отображает данные, не выполняя преобразований в формат HTML-документа
@param	Документирует параметр метода
@provides	Документирует службу, поддерживаемую модулем (добавлено в JDK 9)
@return	Документирует значение, возвращаемое методом

Окончание таблицы

Дескриптор	Описание
`@see`	Определяет ссылку на другую тему
`@serial`	Документирует поле, сериализуемое по умолчанию
`@serialData`	Документирует данные, записываемые методом `writeObject()` или `writeExternal()`
`@serialField`	Документирует компонент `ObjectStreamField`
`@since`	Обозначает версию, в которой были внесены определенные изменения
`@throws`	То же, что и дескриптор `@exception`
`@uses`	Документирует службу, требуемую для модуля (добавлено в JDK 9)
`{@value}`	Отображает значение константы, которая должна быть определена как поле типа `static`
`@version`	Обозначает версию класса

Дескрипторы, начинающиеся с символа `@`, называются ***автономными*** (или ***блочными***) и помечают строку комментариев, тогда как дескрипторы, заключенные в фигурные скобки, такие как `{@code}`, называются ***встраиваемыми*** и могут быть использованы в других дескрипторах. В документирующих комментариях также можно использовать стандартные HTML-дескрипторы. Но некоторые HTML-дескрипторы, например дескрипторы заголовков, применять не следует, поскольку они могут испортить внешний вид HTML-документа, формируемого с помощью утилиты `javadoc`.

Что касается документирования исходного кода, то документирующие комментарии можно использовать для описания классов, интерфейсов, полей, конструкторов и методов. Но в любом случае документирующие комментарии должны предшествовать непосредственно описываемому элементу исходного кода. Одни дескрипторы, в том числе `@see`, `@since` и `@deprecated`, могут быть использованы для документирования любых элементов исходного кода, а другие — только для документирования определенных элементов.

Примечание

Документирующие комментарии также можно использовать для составления краткого обзора разрабатываемого пакета, но делается это иначе, чем в случае документирования исходного кода. Подробнее об этом можно узнать из документации к утилите `javadoc`**. Начиная с JDK 9 утилита** `javadoc` **может также применяться для документирования файла** `module-info.java`.

@author

Описывает автора программного блока и имеет следующий синтаксис:

```
@author описание
```

где описание, как правило, обозначает имя автора. Для того чтобы сведения, указываемые в поле `@author`, были включены в результирующий HTML-документ, при вызове утилиты `javadoc` из командной строки следует указать параметр `-author`.

{@code}

Позволяет включать в комментарии текст, в том числе и отдельные фрагменты кода. Такой текст выводится специальным шрифтом, используемым для форматирования кода, и не подлежит дальнейшей обработке по правилам форматирования HTML-документов. Этот дескриптор имеет следующий синтаксис:

```
{@code фрагмент_кода}
```

@deprecated

Указывает на то, что программный элемент не рекомендован к применению. В описание рекомендуется включать дескриптор `@see` или `{@link}`, чтобы уведомить программиста о других возможных решениях. Этот дескриптор имеется следующий синтаксис:

```
@deprecated описание
```

где *описание* обозначает сообщение, описывающее причины, по которым данное языковое средство Java не рекомендуется к применению. Дескриптор `@deprecated` можно применять для документирования полей, методов, конструкторов, классов и интерфейсов.

{@docRoot}

Указывает путь к корневому каталогу документации.

@exception

Описывает исключение, которое может возникнуть при выполнении метода. Имеет следующий синтаксис:

```
@exception имя_исключения пояснение
```

где *имя_исключения* обозначает полностью определенное имя исключения, а *пояснение* — строку, в которой поясняется, при каких условиях может возникнуть исключение. Дескриптор `@exception` можно применять только для документирования методов и конструкторов.

@hidden

С помощью этого дескриптора предотвращается отображение элемента в документации. Этот дескриптор появился в JDK 9.

{@index}

Указывает на элемент, который будет проиндексирован и, таким образом, найден при использовании функции поиска (появился в JDK 9). Этот дескриптор имеет следующий синтаксис:

```
{@index термин строка_использования }
```

где *термин* — это элемент (который может быть цитируемой строкой), предназначенный для индексации. Параметр *строка_использования* является необязательным. Например, в следующем дескрипторе `@exception` дескриптор `{@index}` добавляет в индекс термин `"error"`:

```
@exception IOException On input {@index error}.
```

Обратите внимание на то, что слово `"error"` отображается как часть описания, но теперь оно еще и индексируется. Если же задать необязательный параметр *строка_использования*, то это описание будет отображаться в индексе и в поле поиска, указывая таким образом, как используется данный термин. Например, дескриптор `{@index error Серьезная ошибка выполнения}` отображает сообщение `"Серьезная ошибка выполнения"`, связанное с термином `"error"`, в индексе и в поле поиска. Этот дескриптор также был добавлен в JDK 9.

{@inheritDoc}

Наследует комментарии от ближайшего суперкласса.

{@link}

Предоставляет встраиваемую ссылку на дополнительные сведения и имеет следующий синтаксис:

```
{@link пакет.класс#член текст}
```

где *пакет.класс#член* обозначает имя класса или метода, на который делается встраиваемая ссылка, а текст — строку, отображаемую в виде встраиваемой ссылки.

{@linkplain}

Вставляет встраиваемую ссылку на другую тему. Эта ссылка отображается обычным шрифтом. В остальном же данный дескриптор подобен дескриптору `{@link}`.

{@literal}

Позволяет включать текст в комментарии. Этот текст отображается без дополнительной обработки по правилам форматирования HTML-документов. Данный дескриптор имеет следующий синтаксис:

```
@literal описание
```

где *описание* обозначает текст, включаемый в комментарии.

@param

Описывает параметр и имеет следующий синтаксис:

```
@parameter имя_параметра пояснение
```

где *имя_параметра* задает конкретное имя параметра, а *пояснение* — его назначение. Дескриптор `@param` можно применять для документирования метода, конструктора, а также обобщенного класса или интерфейса.

@provides

Документирует службу, поддерживаемую модулем, и имеет следующий синтаксис:

```
@provides тип пояснение
```

где *тип* определяет тип провайдера службы, а *пояснение* описывает провайдера службы. Этот дескриптор появился в JDK 9.

@return

Описывает значение, возвращаемое методом, и имеет следующий синтаксис:

```
@return пояснение
```

где *пояснение* обозначает тип и структуру возвращаемого значения. Дескриптор `@return` применяется только для документирования методов.

@see

Предоставляет ссылку на дополнительные сведения. Ниже приведены две наиболее часто используемые формы этого дескриптора.

```
@see ссылка

@see пакет.класс#член текст
```

В первой форме *ссылка* обозначает абсолютный или относительный URL-адрес. Во второй форме *пакет.класс#член* обозначает имя элемента, а *текст* — отображаемые сведения об этом элементе. Параметр *текст* указывать необязательно, а в его отсутствие отображается элемент, определяемый параметром *пакет.класс#член*. Имя члена также может быть опущено. Этот дескриптор дает возможность указать ссылку не только на метод или поле, но и на класс или интерфейс. Имя элемента может быть указано полностью или частично. Но если имени члена предшествует точка, то она должна быть заменена знаком `#`.

@serial

Определяет комментарии к полю, сериализуемому по умолчанию, и имеет следующий синтаксис:

```
@serial описание
```

где *описание* обозначает комментарии к данному полю.

@serialData

Предназначен для документирования данных, которые были записаны с помощью методов `writeObject()` и `writeExternal()`, и имеет следующий синтаксис.

```
@serialData описание
```

где *описание* обозначает комментарии к записанным данным.

@serialField

Предназначен для документирования классов, реализующих интерфейс `Serializable`. Он предоставляет комментарии к компоненту `ObjectStreamField` и имеет следующий синтаксис:

```
@serialField имя тип описание
```

где *имя* и *тип* обозначают конкретное наименование и тип поля соответственно, а *описание* — комментарии к этому полю.

@since

Устанавливает, что данный элемент был внедрен начиная с указанной версии программы. Синтаксис этого дескриптора таков:

```
@since версия
```

Здесь *версия* обозначает строку, указывающую на версию или выпуск программы, когда был внедрен данный элемент.

@throws

Выполняет те же функции, что и дескриптор `@exception`.

@uses

Документирует провайдера службы, требуемого для модуля, и имеет следующий синтаксис:

```
@uses тип пояснение
```

где параметр *тип* задает провайдера службы, а параметр *пояснение* описывает службу. Этот дескриптор появился в JDK 9.

{@value}

Применяется в двух основных формах. В первой форме отображается значение константы, которой предшествует этот дескриптор. Константа должна быть полем типа `static`.

```
{@value}
```

Во второй форме отображается значение указываемого статического поля:

```
{@value пакет.класс#член }
```

где *пакет.класс#член* обозначает имя статического поля.

@version

Описывает версию программного элемента и имеет такой синтаксис:

```
@version информация
```

где *информация* обозначает строку, содержащую сведения о версии программы. Как правило, это номер версии, например 2.2. Для того чтобы сведения в поле дескриптора `@version` были включены в результирующий HTML-документ, при вызове утилиты `javadoc` из командной строки следует указать параметр `-version`.

Общая форма документирующих комментариев

После символов `/**` следуют одна или несколько строк с общим описанием класса, интерфейса, переменной, конструктора, метода или модуля. Далее можно ввести произвольное количество дескрипторов, начинающихся со знака `@`. Каждый такой дескриптор должен начинаться с новой строки или следовать после одной или нескольких звездочек (`*`) в начале строки. Несколько однотипных дескрипторов должны быть объединены вместе. Так, если требуется использовать три дескриптора `@see`, их следует расположить друг за другом. Встраиваемые дескрипторы (начинающиеся с фигурной скобки) можно использовать в любом описании.

Ниже приведен пример, демонстрирующий использование документирующих комментариев для описания класса.

```
/**
* Класс для отображения гистограммы.
* @author Herbert Schildt
* @version 3.2
*/
```

Результат, выводимый утилитой javadoc

Утилита `javadoc` читает данные из исходного файла программы на Java и генерирует несколько HTML-файлов, содержащих документацию к этой программе. Сведения о каждом классе помещаются в отдельный файл. В результате выполнения утилиты `javadoc` создается также предметный указатель (индекс) и дерево иерархии. Кроме того, могут быть сгенерированы и другие HTML-файлы.

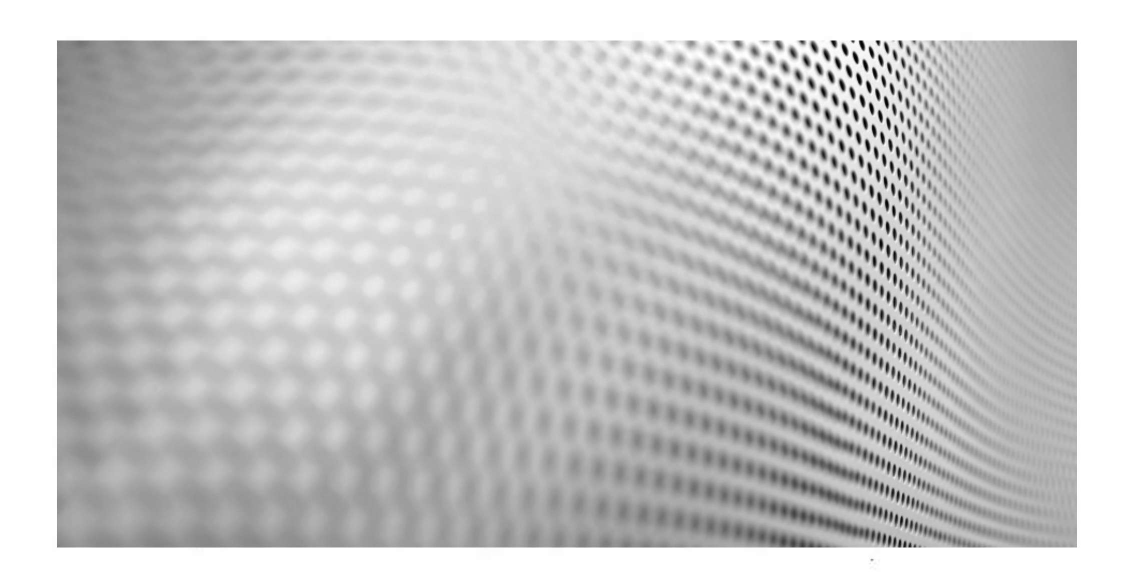
Пример использования документирующих комментариев

Ниже приведен пример программы, в исходном тексте которой имеются документирующие комментарии. Обратите внимание на то, что каждый такой комментарий непосредственно предшествует описываемому элементу

программы. После обработки утилитой javadoc документация по классу SquareNum помещается в файл SquareNum.html.

```
import java.io.*;

/**
   * Класс, демонстрирующий использование
   * документирующих комментариев.
   * @author Herbert Schildt
   * @version 1.2
   */

public class SquareNum {
   /**
   * Этот метод возвращает квадрат значения параметра num.
   * Данное описание состоит из нескольких строк. Число строк
   * не ограничивается.
   * @param num Значение, которое требуется возвести в квадрат.
   * @return Квадрат числового значения параметра num.
   */
   public double square(double num) {
      return num * num;
   }

   /**
   * Этот метод получает значение, введенное пользователем.
   * @return Введенное значение типа double.
   * @exception IOException Исключение при ошибке ввода.
   * @see IOException
   */
   public double getNumber() throws IOException {
      // Создать поток BufferedReader из стандартного
      // потока System.in
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader inData = new BufferedReader(isr);
      String str;

      str = inData.readLine();
      return (new Double(str)).doubleValue();
   }

   /**
   * В этом методе демонстрируется применение метода square().
   * @param args Не используется.
   * @exception IOException Исключение при ошибке ввода.
   * @see IOException
   */
   public static void main(String args[]) throws IOException
   {
      SquareNum ob = new SquareNum();
      double val;
```

```
        System.out.println("Введите значение для возведения
                            в квадрат: ");
        val = ob.getNumber();
        val = ob.square(val);

        System.out.println("Квадрат введенного значения " + val);
    }
}
```



Приложение В

Обзор технологии Java Web Start

Как упоминалось в главе 1, с выходом JDK 9 аплеты не рекомендуется применять для разработки веб-приложений. Несмотря на то что аплеты использовались в Java на протяжении многих лет, у них есть свои недостатки. Один из них заключается в том, что они задействуют подключаемый модуль браузера, поддержка которого постепенно прекращается. В силу этой и ряда других причин для разработки веб-приложений рекомендуется применять технологию Java Web Start, для которой подключаемый модуль браузера не требуется. Соответственно, приложения, развертываемые с помощью Java Web Start, могут выполняться независимо от браузера.

Важно отметить, что тема, связанная с Java Web Start и стратегиями развертывания, довольно обширна. Более того, в стратегиях развертывания задействуется множество других технологий, таких как JAR-файлы, файлы манифеста, сертификаты приложений и JNLP-файлы. Также существует немало соображений, связанных с развертыванием коммерческих приложений. Поэтому подробное обсуждение стратегий развертывания выходит за рамки данной книги. Но в силу растущей важности Java Web Start необходимо дать хотя бы краткий обзор этой технологии, чтобы вы получили общее представление о ее возможностях.

Примечание

Поскольку механизмы развертывания приложений Java постоянно меняются и совершенствуются, особенно в том, что касается безопасности, за последними сведениями рекомендуется обратиться к документации Oracle. Также предполагается, что на вашем компьютере установлена современная версия Java.

Знакомство с Java Web Start

По сути, Java Web Start (JavaWS) — это механизм, поддерживающий развертывание веб-приложений Java. В отличие от аплета Java, который должен расширять класс `Applet` или `JApplet` и поддерживать общую архитектуру аплета с помощью методов `init()`, `start()`, `stop()` и `destroy()`, приложения JavaWS являются обычными клиентскими программами, подобными приложениям Swing и JavaFX, которые рассматривались в соответствующих главах книги. То есть это полнофункциональные клиентские приложения, которые могут загружаться и выполняться из Интернета. Например, такая программа, как `ButtonDemo` (см. главу 16), может выполняться как приложение JavaWS без всяких изменений.

В связи с тем что Java Web Start не использует подключаемый модуль браузера, для выполнения приложения JavaWS на главном компьютере нужно установить лишь среду JRE. В результате исчезает проблема отсутствующего, отключенного или устаревшего браузерного модуля Java. И поскольку приложение

JavaWS выполняется на рабочем столе (а не в окне браузера), ваша программа выглядит как стандартное оконное приложение. Более того, как только приложение JavaWS будет загружено, оно сможет выполняться в автономном режиме. Можно также создать ярлык на такое приложение. Фактически Java Web Start предоставляет разработчикам возможность создавать приложения, которые обладают намного более широкими функциональными возможностями, чем аплеты.

По умолчанию приложения JavaWS запускаются в той же защищенной “песочнице”, которая используется неподписанными аплетами, поэтому они имеют те же самые ограничения. Таким образом, изначально они обеспечивают уровень безопасности, аналогичный неподписанным аплетам. Но при необходимости им можно предоставить дополнительные полномочия.

Развертывание Java Web Start

Несмотря на наличие множества нюансов, связанных с развертыванием Java Web Start, есть четыре этапа, которые следует считать ключевыми. Во-первых, приложение JavaWS должно быть упаковано в JAR-файл. Во-вторых, JAR-файл должен быть подписан. В-третьих, должен быть создан файл JNLP, содержащий информацию, которая требуется для запуска приложения. И наконец, как правило, нужно создать ссылку на файл JNLP, который служит для запуска приложения. Каждый из этих этапов вкратце описан в следующих разделах.

Для приложений JavaWS требуется JAR-файл

Все приложения JavaWS нужно упаковать в файл JAR. Как уже упоминалось, аббревиатура JAR расшифровывается как Java Archive (архив Java). Файл JAR создается с помощью утилиты командной строки `jar`. При создании JAR-файла для Java Web Start нужно указать все файлы, включая классы и ресурсы, используемые приложением, а также информацию, которая будет включена в манифест приложения. Манифест содержит сведения о файлах JAR, включая настройки безопасности.

Утилита `jar` поддерживает множество опций, но для создания простых приложений потребуются только три из них: `c`, `f` и `m`. Опция `c` означает создание архива, `f` задает имя архива, а `m` указывает на необходимость включения информации в файл манифеста. Например, в результате выполнения следующей команды создается JAR-файл под названием `MyJar.jar`, который включает класс из файла `MyClass.class`, а также информацию манифеста, заданную в файле `MyMan.txt`:

```
jar cfm MyJar.jar MyMan.txt MyClass.class
```

Обратите внимание на то, что имена файлов перечисляются в том же порядке, что и опции.

Подписанные приложения JavaWS

В общем случае приложение JavaWS должно быть подписано действительным сертификатом. При этом фактически подписывается JAR-файл приложения. Говоря простым языком, сертификат идентифицирует владельца приложения. А поскольку подписанные JAR-файлы становятся недействительными в случае изменения, тем самым обеспечивается целостность файлов, связанных с приложением. Сертификат должен быть получен у авторизованного независимого центра сертификации. Обычно подобные сертификаты выдаются на платной основе.

Прежде чем двигаться дальше, рассмотрим специальный тип сертификата, который называется "самозаверяющим". Такой сертификат создается пользователем, а не выдается центром сертификации. На момент написания книги приложения JavaWS можно было подписывать с помощью самозаверяющего сертификата. Важно понимать, что в современных версиях Java самозаверяющее приложение невозможно запустить на выполнение до тех пор, пока оно не будет добавлено в список исключений на панели управления Java. Но даже в этом случае все равно отобразится запрос системы безопасности при попытке запустить приложение. Как следствие, самозаверяющие приложения не могут применяться для развертывания. Это особенно важно для коммерческих приложений. Все коммерческие приложения JavaWS должны быть подписаны с помощью действительного признанного сертификата. Тем не менее в некоторых случаях самозаверение может быть полезным при изучении технологии Java Web Start, а также при разработке и отладке приложений JavaWS.

Для подписания JAR-файлов применяется утилита командной строки `jarsigner`. Но для ее использования нужен сертификат. Как уже упоминалось, для общего развертывания приложения, особенно в случае коммерческого кода, следует использовать сертификат от независимого центра сертификации. Однако для целей изучения или экспериментирования можно получить самозаверяющий сертификат с помощью утилиты `keytool`. В Java цифровые подписи основаны на механизме безопасности с использованием открытого/закрытого ключей. Утилита `keytool` работает с файлом *хранилища ключей*, который содержит ключи и управляет сертификатами. В приводимом далее примере будет показано использование утилит `jarsigner` и `keytool` для самозаверения демонстрационной программы.

Приложение

На момент написания книги было технически возможно использовать неподписанный JAR-файл при работе с приложением JavaWS, если файл JNLP находится в списке исключений на панели управления Java. Разумеется, настоятельно не рекомендуется развертывать неподписанные приложения.

Использование файлов JNLP при работе с приложениями JavaWS

Приложение JavaWS запускается с помощью файла JNLP (Java Network Launch Protocol — сетевой протокол запуска приложений Java). Файл JNLP фактически является файлом XML, который использует элемент jnlp и имеет расширение .jnlp. Элемент jnlp включает описание приложения JavaWS. Несмотря на то что элемент jnlp поддерживает множество опций, для простых приложений используются лишь некоторые из них. Обычно задается общее описание приложения и указываются используемые им ресурсы. Вот пример простого файла JNLP, применяемого для запуска программы ButtonDemo, которая рассматривалась в главе 16.

```
<?xml version="1.0" encoding="UTF-8"?>

<jnlp href="ButtonDemo.jnlp" spec="6.0+">

   <!-- Описание приложения -->
   <application-desc main-class="ButtonDemo">
   </application-desc>

   <!-- Ресурсы, требуемые приложению -->
   <resources>
      <!-- Здесь дается ссылка на jar-файл -->
      <jar href="ButtonDemo.jar" main="true"/>

      <!-- Определение минимальной версии Java -->
      <java version="1.8+"/>
   </resources>

   <!-- Информация, относящаяся к приложению -->
   <information>
      <!-- Эти сведения отображаются в списке кэша
           панели управления Java -->
      <title>ButtonDemo Application</title>
      <vendor>Self</vendor>

      <!-- Разрешается запуск в автономном режиме -->
      <offline-allowed/>
   </information>

</jnlp>
```

Давайте детальнее изучим содержимое этого файла. В первой строке указывается минимальная версия XML и кодировка UTF. Это не обязательное требование, а лишь рекомендация. Здесь используются типичные значения, которые можно изменять в соответствии с потребностями конкретного приложения. Атрибут href элемента jnlp определяет имя файла JNLP. С помощью атрибута spec задается минимальная версия JNLP. В рассматриваемом примере выбрана версия 6.0 или выше.

Для описания приложения используется элемент `application-desc`. Обратите внимание на то, что в атрибуте `main-class` указывается главный класс приложения. Ну а элемент `resources` служит для идентификации ресурсов приложения. В рассматриваемом примере используемый JAR-файл указывается с помощью элемента `jar`. Минимальная версия Java, требуемая для запуска приложения, задается с помощью элемента `java`. (В версиях JNLP до 6 этот элемент назывался `j2se`, причем это имя по-прежнему разрешено.) В данном случае в качестве минимальной версии указана Java 8, но это значение можно изменить. Вообще рекомендуется использовать наименьший номер версии, в которой может выполняться приложение.

Информация, относящаяся к приложению, определяется элементом `information`. Здесь указываются название и поставщик приложения. Также обратите внимание на то, что благодаря включению элемента `offline-allowed` приложение может выполняться в автономном режиме. Это полезно при использовании программ, которые не требуют подключения к Интернету.

Следует отметить, что элемент `jnlp` поддерживает больше опций и атрибутов, чем тут показано. При желании изучите их самостоятельно. Файл JNLP можно настроить максимально точно в соответствии с конкретными особенностями приложения.

И последнее. Несмотря на то что файл JNLP применяется для запуска приложения из браузера, само приложение выполняется вне браузера. Это исключает необходимость в использовании подключаемого модуля Java, что является одним из ключевых преимуществ Java Web Start.

Связывание с файлом JNLP

В общем случае можно создать ссылку на веб-страницу, которая запускает приложение. Вот простейший пример.

```
<body>
<a href="jnlp-path">Launch App</a>
</body>
```

Здесь *`jnlp-path`* задает путь к JNLP-файлу приложения, которое будет запускаться. Вместо *`jnlp-path`* подставьте фактический путь к файлу. После щелчка на этой ссылке браузер перенаправляется к соответствующему файлу JNLP, и запускается приложение JavaWS. Программа будет выполняться на рабочем столе, как объяснялось ранее. Можно просто оставить страницу со ссылкой открытой, и программа будет оставаться активной до тех пор, пока вы не закроете страницу.

Эксперименты с Java Web Start в локальной файловой системе

В этом разделе мы создадим простое приложение JavaWS, с которым можно проводить различные эксперименты. Как уже отмечалось, приложения JavaWS обычно загружаются из Интернета. Но исходя из предположения, что используемый вами браузер позволяет открывать локальные файлы, поэкспериментируем с Java Web Start, запустив приложение из файла на диске. В таком случае вы сможете увидеть, как работает Java Web Start без обращения к веб-серверу. Вы также сможете запустить приложение подобно рядовому пользователю, путем щелчка на ссылке, находящейся на веб-странице.

Как уже упоминалось, программы, развертываемые в Java Web Start, фактически являются обычными приложениями Java. Поэтому совместно с Java Web Start можно применять приложения Swing и JavaFX. В этом примере мы будем использовать Swing-программу `ButtonDemo`, которая была создана в главе 16.

Чтобы создать и развернуть приложение `ButtonDemo` из файла на вашем компьютере, выполните следующие действия.

1. Создайте JAR-файл для приложения `ButtonDemo` под названием `ButtonDemo.jar`.
2. Создайте хранилище ключей, содержащее самозаверяющий сертификат, и подпишите файл `ButtonDemo.jar`.
3. Создайте файл JNLP под названием `ButtonDemo.jnlp`, который описывает и запускает приложение `ButtonDemo`.
4. Создайте короткий HTML-файл под названием `StartBD.html`, который содержит ссылку на файл `ButtonDemo.jnlp`.
5. Добавьте файл `ButtonDemo.jnlp` в список исключений на панели управления Java.
6. В окне браузера откройте файл `StartBD.html` и щелкните на ссылке. После завершения проверки безопасности на рабочем столе запустится приложение `ButtonDemo` (не в окне браузера, как это было в случае аплетов).

Этапы этой пошаговой инструкции подробно рассматриваются в следующих разделах.

Следует отметить, что далеко не все этапы необходимы для выполнения приложения JavaWS в локальной файловой системе. Например, на момент написания книги самозаверение приложения было необязательным. Фактически самозаверение упоминается здесь лишь для демонстрации методики подписания JAR-файла. Также можно обойтись без HTML-файла, поскольку можно непосредственно запускать файл JNLP на выполнение. В данном случае все этапы нужны лишь для лучшего ознакомления с процессом. Кроме того, в будущем могут потребоваться дополнительные этапы.

Создание JAR-файла для приложения ButtonDemo

Начните с компиляции файла ButtonDemo.java, код которого приведен в главе 16 (разумеется, если вы этого не сделали ранее). В результате будут созданы два файла классов: ButtonDemo.class и ButtonDemo$1.class. В первом файле содержится главный класс приложения, а во втором находится выполняемый экземпляр, созданный путем вызова метода SwingUtilities.invokeLater(). Для выполнения приложения нужны оба этих класса, поэтому их следует включить в JAR-файл.

Чтобы создать JAR-файл для приложения, воспользуйтесь утилитой jar. Вот соответствующая команда для приложения ButtonDemo.

```
jar cfm ButtonDemo.jar MyMan.txt ButtonDemo.class ButtonDemo$1.class
```

Опция c определяет создание JAR-файла. Опция f задает имя файла, в данном случае ButtonDemo.jar. Опция m задает включение информации из файла MyMan.txt в манифест приложения, связанный с JAR-файлом.

Манифест JAR-файла содержит информацию, относящуюся к приложению. Все JAR-файлы включают манифест, в который добавляется информация из текстового файла, указанного с помощью опции m. Для приложения ButtonDemo создайте файл MyMan.txt следующего вида:

```
Main-Class: ButtonDemo
Permissions: sandbox
```

В этом коде указано, что главный класс программы называется ButtonDemo, а выполнение программы ограничено "песочницей", которая представляет собой наиболее ограничивающую настройку Java. Эта настройка должна соответствовать тому, что указано в JNLP-файле приложения.

Создание хранилища ключей и подписание файла ButtonDemo.jar

Для всех современных версий Java приложение JavaWS должно подписываться с помощью действительного сертификата безопасности. Сертификат идентифицирует владельца приложения, а благодаря подписанию гарантируется целостность соответствующего JAR-файла. Сертификаты Java основаны на использовании механизма безопасности с применением открытого/закрытого ключей. Эти ключи хранятся в специальном файле, который называется *хранилищем ключей*. Перед подписанием приложения нужно создать хранилище ключей, в котором находится сертификат. Хранилища ключей и сертификаты управляются с помощью утилиты keytool, которая входит в JDK.

Как объяснялось ранее, для общего развертывания потребуется сертификат, полученный от авторизованного независимого центра сертификации. Но для получения представления о процессе подписания можно воспользоваться самозаверяющим сертификатом. Помните о том, что самозаверяющее приложение не подходит для распространения, потому что оно вызовет появление

предупреждения безопасности, которое воспрепятствует запуску вашего приложения. Такое приложение может быть полностью заблокировано. Но зато благодаря использованию самозаверяющего сертификата вы сможете совершенно бесплатно познакомиться с процессом подписания JAR-файла.

Как и следовало ожидать, утилита `keytool` поддерживает множество опций, но лишь некоторые из них требуются в данном примере. Вот один из вариантов:

```
keytool -genkeypair -alias devName -keystore devKeys
```

Опция `-genkeypair` указывает на необходимость генерирования новой пары ключей и создания самозаверяющего сертификата для этой пары. Имя, указываемое после опции `-alias`, в данном случае `devName`, идентифицирует запись в хранилище ключей. Имя хранилища ключей задается опцией `-keystore`, в данном случае это `devKeys`.

После ввода команды вам будет предложено указать следующую информацию: пароль, ваше имя, название подразделения, название организации, город, область и код страны (для России это RU). Затем эта команда автоматически создает самозаверяющий сертификат на основе предоставленной вами информации. Убедитесь в том, что запомнили пароль, поскольку он вам пригодится на следующем этапе.

Примечание

Версии утилиты `keytool` до Java 6 требовали использования опции `-selfcert` для генерирования самозаверяющего сертификата.

Теперь, когда у вас есть сертификат, воспользуйтесь утилитой `jarsigner` для подписания файла `ButtonDemo.jar`. Эта утилита также поддерживает множество опций, но в данном примере требуется лишь одна из них, которая задает хранилище ключей:

```
jarsigner -keystore devKeys ButtonDemo.jar devName
```

В данном случае опция `-keystore` определяет файл хранилища ключей, который в нашем примере называется `devKeys`. Далее указывается имя подписываемого JAR-файла, в данном случае `ButtonDemo.jar`. И замыкает всю эту цепочку псевдоним сертификата. После выполнения этой команды появится запрос на ввод пароля, который был задан на предыдущем этапе. Не забывайте о том, что в случае изменения файла `ButtonDemo.jar` его понадобится подписать повторно.

Примечание

Не забывайте о том, что хотя самозаверение может быть полезным при демонстрации процедуры подписания JAR-файлов, самозаверяющий сертификат не следует использовать при развертывании реальных приложений. В этом случае понадобится сертификат,

полученный от авторизованного центра сертификации. Более того, самозаверяющее приложение несет угрозу безопасности. Поэтому, как правило, нужно проявлять большую осторожность, прежде чем пытаться запускать самозаверяющие приложения, которые вы не создавали.

Создание файла JNLP для приложения ButtonDemo

На следующем этапе нужно создать файл JNLP, который запускает приложение ButtonDemo. Этот файл вы уже видели раньше. Несмотря на его простоту, для данного примера его вполне достаточно. Назовем этот файл ButtonDemo.jnlp.

```
<?xml version="1.0" encoding="UTF-8"?>

<jnlp href="ButtonDemo.jnlp" spec="6.0+">

   <!-- Описание приложения -->
   <application-desc main-class="ButtonDemo">
   </application-desc>

   <!-- Ресурсы, требуемые приложению -->
   <resources>
      <!-- Здесь дается ссылка на jar-файл -->
      <jar href="ButtonDemo.jar" main="true"/>

      <!-- Определение минимальной версии Java -->
      <java version="1.8+"/>
   </resources>

   <!-- Информация, относящаяся к приложению -->
   <information>
      <!-- Эти сведения отображаются в списке кэша
           панели управления Java -->
      <title>ButtonDemo Application</title>
      <vendor>Self</vendor>

      <!-- Разрешается запуск в автономном режиме -->
      <offline-allowed/>
   </information>

</jnlp>
```

Создание HTML-файла StartBD.html

Несмотря на то что зачастую можно непосредственно запускать JNLP-файл, скачиваемый из Интернета, обычно для запуска приложения предоставляется ссылка. Вот простой пример ссылки, применяемой для запуска приложения ButtonDemo.

```
<body>
<a href="ButtonDemo.jnlp">Запуск приложения ButtonDemo</a>
</body>
```

Назовите этот файл StartBD.html. При щелчке на ссылке браузер перенаправляется к файлу ButtonDemo.jnlp, после чего с помощью Java Web Start запускается программа. Она будет выполняться на рабочем столе, как объяснялось ранее. Можно просто оставить страницу со ссылкой открытой, и программа будет оставаться активной до тех пор, пока вы не закроете страницу.

Добавление файла ButtonDemo.jnlp в список исключений на панели управления Java

Поскольку файл ButtonDemo.jar является самозаверяющим и вызывается из локальной файловой системы, добавьте URL-адрес файла ButtonDemo.jnlp в список исключений на панели управления Java. (Этот список появился в Java 7, обновление 51.) Предполагая, что файл находится в папке C:\Java\MyDevFiles, добавьте в список исключений следующую строку:

```
FILE:/C:\Java\MyDevFiles\ButtonDemo.jnlp
```

Здесь FILE: обозначает файл, находящийся в локальной файловой системе. Чтобы выполнить стандартное развертывание приложения, нужно указать сетевой адрес.

Примечание

Как правило, появление в списке исключений URL-адреса, использующего префикс FILE: (т.е. ссылка на файл в локальной файловой системе), представляет собой угрозу безопасности. Поэтому хорошенько подумайте, прежде чем добавлять такой URL-адрес в список исключений для программы, которую вы не создавали.

Выполнение приложения ButtonDemo в браузере

Теперь можно запустить приложение ButtonDemo в браузере. Для этого откройте файл StartBD.html в окне браузера. Например, в Internet Explorer для открытия файла выберите команду Открыть в меню Файл. (Как уже упоминалось в начале раздела, данный пример требует, чтобы браузер позволял открывать файлы в локальной файловой системе.) Как только отобразится веб-страница, щелкните на ссылке Запуск приложения ButtonDemo. Даже если вы добавите файл ButtonDemo.jnlp в список исключений, все равно при первом запуске программы может появиться предупреждение системы безопасности. Ответьте утвердительно, чтобы запустить приложение ButtonDemo на выполнение. Все будет выглядеть так, как будто оно запускается в обычном режиме из командной строки (см. главу 16).

Примечание

Не забывайте о том, что запуск приложения JavaWS из файла в локальной файловой системе и использование самозаверяющего сертификата разрешены только для

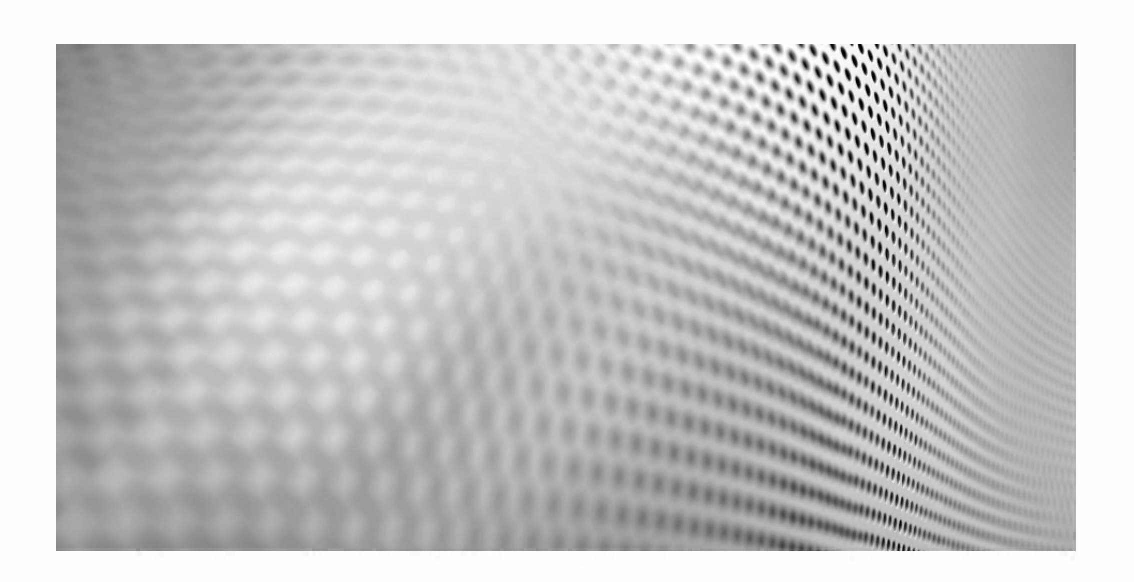
отладки и в учебных целях. Если же речь идет о фактическом развертывании приложения, потребуется получить сертификат от авторизованного центра сертификации, а само приложение будет развернуто по сети.

Запуск приложения JavaWS с помощью утилиты `javaws`

При разработке приложений JavaWS вовсе не обязательно запускать приложения с помощью браузера. Вместо этого можно использовать утилиту `javaws` для запуска приложения непосредственно из командной строки. Для этого просто укажите имя файла JNLP. Например, предположим, что файлы `ButtonDemo.jnlp` и `ButtonDemo.jar` находятся в текущей рабочей папке. Тогда команда `javaws ButtonDemo.jnlp` запускает программу `ButtonDemo` без использования браузера или HTML-файла. Это позволяет ускорить циклы компиляции/тестирования/отладки. Разумеется, при этом следует учитывать ограничения безопасности.

Использование Java Web Start для запуска аплетов

Несмотря на то что аплеты уже выходят из употребления, они по-прежнему поддерживаются в Java Web Start и могут запускаться файлом JNLP. Подобный подход можно увидеть в примерах унаследованного кода. Элемент, описывающий аплет, называется `applet-desc`. При обновлении устаревшего кода нужно преобразовать аплет в приложение и описать его с помощью элемента `application-desc`, как было показано ранее.



Приложение Г

Введение в JShell

В версии JDK 9 появилась утилита JShell, поддерживающая интерактивную среду, которая позволяет быстро и легко экспериментировать с кодом Java. Благодаря JShell реализуется то, что называется циклом *чтение — вычисление — вывод* (read-evaluate-print loop, REPL). Эта утилита запрашивает у пользователя фрагмент кода, который считывается и вычисляется, после чего отображается результат работы кода, например вывод, сгенерированный методом `println()`, результат выражения или текущее значение переменной. Далее JShell выводит очередной запрос, и процесс повторяется. В терминологии JShell каждая вводимая последовательность кода называется *фрагментом*.

Ключом к пониманию JShell является то, что вам не нужно вводить завершенную Java-программу. Оценивание каждого фрагмента кода выполняется в процессе его ввода. JShell автоматически обрабатывает многие детали, связанные с функционированием Java. Это позволяет концентрироваться на конкретных деталях без необходимости писать полноценную программу. Это делает JShell особенно удобным инструментом для тех, кто только начинает изучать Java.

В то же время JShell может оказаться достаточно полезным средством и для опытных программистов. Поскольку JShell сохраняет информацию о состоянии, можно вводить многострочные последовательности кода и выполнять их в JShell. Это позволяет оперативно проверять различные программные идеи, проводя интерактивные эксперименты с кодом, но не компилируя итоговую программу.

В этом приложении вы ознакомитесь с интерпретатором JShell и освоите его ключевые средства, особенно полезные для начинающих программистов на Java.

Основы JShell

JShell является утилитой командной строки. Для запуска сеанса JShell в командной строке введите **`jshell`**. В результате появится приглашение интерпретатора команд:

```
jshell>
```

В ответ на это приглашение можно ввести фрагмент кода или команду JShell.

JShell позволяет ввести отдельную инструкцию и сразу же видеть результат ее выполнения. Вернемся к первому примеру Java-программы в этой книге. Вот как он выглядит.

```
class Example {
   // Выполнение программы на Java начинается с вызова метода main()
   public static void main(String args[]) {
      System.out.println("Java правит Интернетом!");
   }
}
```

В этой программе реальные действия выполняет лишь метод `println()`, который выводит сообщение на экране. Остальной код формирует объявление класса и метода. В JShell не требуется явно указывать класс или метод для

вызова метода `println()`. Интерпретатор способен непосредственно выполнить его. Чтобы убедиться в этом, введите в командной строке приглашения JShell следующую инструкцию и нажмите клавишу <Enter>:

```
System.out.println("Java правит Интернетом!");
```

Результат будет выглядеть следующим образом.

```
Java правит Интернетом!

jshell>
```

Как видите, интерпретатор проанализировал вызов метода `println()` и вывел его строковый аргумент, после чего снова отображается приглашение.

Прежде чем двигаться дальше, будет полезно объяснить, почему интерпретатор JShell способен выполнить одиночную инструкцию, такую как вызов метода `println()`, в то время как компилятор Java, `javac`, требует завершенную программу. JShell автоматически формирует программную среду (в фоновом режиме), которая состоит из *синтетического класса* и *синтетического метода*. В данном случае вызов `println()` внедряется в синтетический метод, который является частью синтетического класса. В результате введенный код оказывается частью корректной Java-программы, пусть даже вы ее и не видите. Подобный подход обеспечивает очень быстрый и удобный способ экспериментирования с кодом Java.

Теперь рассмотрим, как поддерживаются переменные. В JShell можно объявить переменную, присвоить ей значение и использовать ее в любых допустимых выражениях. Например, введите в командной строке следующую инструкцию:

```
int count;
```

Реакция интерпретатора будет такой:

```
count ==> 0
```

Это означает, что переменная `count` была добавлена в качестве статической переменной в синтетический класс и инициализирована нулевым значением.

Присвоим переменной `count` значение 10 с помощью следующей инструкции:

```
count = 10;
```

Результат будет таким:

```
count ==> 10
```

Как видите, переменной `count` присвоено значение `10`. Поскольку она является статической, ее можно использовать без ссылки на объект.

Поскольку переменная `count` объявлена, она может входить в состав выражения. Например, введите следующую инструкцию:

```
System.out.println("Обратная величина count: " + 1.0 / count);
```

JShell ответит следующим образом:

```
Обратная величина count: 0.1
```

В данном случае результат выражения `1.0 / count` равен `0.1`, поскольку переменной `count` было предварительно присвоено значение `10`.

Этот пример иллюстрирует другой важный аспект JShell: интерпретатор поддерживает информацию о состоянии. В нашем случае переменной `count` присваивается значение 10, и затем данное значение используется в выражении `1.0 / count` в вызове метода `println()`. Между этими двумя инструкциями интерпретатор сохраняет значение переменной `count`. JShell запоминает текущее состояние и результаты работы вводимых фрагментов кода, что позволяет экспериментировать с крупными фрагментами кода, состоящими из нескольких строк.

Рассмотрим еще один пример. На этот раз создадим цикл `for`, в котором используется переменная `count`. Для начала введите следующую строку кода:

```
for(count = 0; count < 5; count++)
```

JShell отреагирует следующим образом:

```
...>
```

Это означает, что для завершения инструкции требуется дополнительный код. В нашем случае следует указать тело цикла `for`. Введите следующую строку:

```
System.out.println(count);
```

После ввода этой строки кода инструкция `for` завершается, и выполняется весь цикл. Результат будет таким.

```
0
1
2
3
4
```

Помимо инструкций и переменных интерпретатор JShell позволяет объявлять классы и методы, а также использовать инструкции импорта. Соответствующие примеры приведены в следующих разделах. И еще один момент: любой код, действительный для JShell, также будет действителен для компиляции с помощью `javac`, при условии, что предоставлена инфраструктура, требуемая для создания полноценной программы. Таким образом, если фрагмент кода может быть выполнен в JShell, то он представляет собой корректный Java-код. Другими словами, код JShell — это код Java.

Вывод, редактирование и повторное выполнение кода

В интерпретаторе JShell поддерживается большое количество команд, позволяющих управлять его работой. Особый интерес среди них представляют три команды, которые предназначены для отображения введенного кода, редактирования строки кода и повторного выполнения фрагмента кода. По мере увеличения последующих фрагментов кода вы сможете в полной мере оценить пользу от этих команд.

В JShell все команды начинаются с символа /, за которым следует команда. Чаще всего используется команда /list, которая отображает ранее введенный код. Предполагая, что вы ввели коды всех примеров, показанных в предыдущем разделе, попробуйте просмотреть их, выполнив команду /list. В ответ на это интерпретатор JShell выдаст пронумерованный список введенных фрагментов кода. Обратите внимание на запись, которая соответствует циклу for. Несмотря на то что цикл состоит из двух строк, они образуют одну инструкцию, которой соответствует один номер фрагмента.

Для редактирования фрагмента кода предназначена команда /edit. В результате ее выполнения открывается окно редактирования, в котором можно изменять код. Существуют три способа вызова данной команды. Во-первых, если ввести команду /edit саму по себе, в окне редактирования отобразятся все введенные строки кода. Во-вторых, можно указать конкретный фрагмент кода для редактирования, выполнив команду /edit *n*, где *n* задает номер фрагмента кода. Например, для редактирования фрагмента 3 используйте команду /edit 3. И наконец, можно указать именованный элемент (скажем, переменную). К примеру, для изменения значения переменной count введите команду /edit count.

Интерпретатор JShell способен не только выполнять код при вводе, но и повторно выполнять ранее введенный код. Для повторного выполнения последнего введенного фрагмента используйте команду /!. Чтобы повторно выполнить конкретный фрагмент, укажите его номер с помощью команды /*n*, где *n* соответствует номеру фрагмента. Например, для повторного выполнения четвертого фрагмента воспользуйтесь командой /4. Можно также задать фрагмент по его позиции относительно текущего фрагмента, указав отрицательное смещение. Скажем, для повторного выполнения фрагмента, который находится на три позиции раньше текущего фрагмента, введите команду /-3.

Еще одна важная команда, о которой нужно знать, — /exit. Она завершает работу интерпретатора JShell.

Добавление метода

Впервые вы узнали о методах в главе 4. Все методы находятся в классах. Но при использовании JShell можно экспериментировать с методами без их *явного* объявления в классе. Как объяснялось выше, это связано с тем, что JShell автоматически обертывает фрагменты кода в синтетический класс. В результате можно быстро написать метод, не создавая инфраструктуру класса. Можно даже вызвать метод, не создавая его объект. Эта возможность JShell особенно полезна при изучении методов в Java либо при экспериментировании с новым кодом. Чтобы лучше понять данный процесс, рассмотрим соответствующий пример.

Для начала откройте новый сеанс JShell и в командной строке введите следующий метод.

```
double reciprocal(double val) {
   return 1.0/val;
}
```

Этот код создает метод, который возвращает обратное значение аргумента. После ввода кода интерпретатор JShell ответит следующим образом:

```
|   created method reciprocal(double)
```

Данное сообщение означает, что в синтетический класс JShell добавлен метод, который готов к применению.

Чтобы вызвать метод `reciprocal()`, просто укажите его имя без какой-либо ссылки на объект или класс, например:

```
System.out.println(reciprocal(4.0));
```

В ответ JShell выведет значение `0.25`.

Вы, наверное, удивитесь, как можно вызвать метод `reciprocal()` без использования оператора точки и ссылки на объект. Причина заключается в том, что при создании автономного метода в JShell, такого как `reciprocal()`, интерпретатор автоматически делает этот метод статическим членом синтетического класса. Как объяснялось в главе 5, статические методы вызываются из класса, а не из конкретного объекта, поэтому объект не требуется. Аналогичным образом автономные переменные становятся статическими переменными синтетического класса, как описывалось ранее.

Еще один важный аспект JShell заключается в поддержке ***прямых ссылок*** в методах. Благодаря этому один метод может вызывать другой, даже если тот, второй, метод еще не определен. Это позволяет вводить метод, который зависит от другого метода, не беспокоясь о том, какой метод был введен первым. Рассмотрим простой пример. Введите следующую строку кода в JShell:

```
void myMeth() { myMeth2(); }
```

В ответ JShell отобразит следующее сообщение.

```
|   created method myMeth(), however, it cannot be invoked
    until myMeth2() is declared
```

Как видите, интерпретатор в курсе того, что метод `myMeth2()` еще не объявлен, и тем не менее дает возможность определить метод `myMeth()`. Как и следовало ожидать, если попытаться вызвать метод `myMeth()`, появится сообщение об ошибке, поскольку метод `myMeth2()` еще не определен, зато можно продолжать вводить код для метода `myMeth()`.

Определим метод `myMeth2()`:

```
void myMeth2() { System.out.println("JShell могуч."); }
```

Вот теперь, когда метод `myMeth2()` определен, можно вызывать метод `myMeth()`.

Прямую ссылку можно использовать не только в методе, но и в инициализаторе поля в классе.

Создание класса

Несмотря на то что JShell автоматически поддерживает синтетический класс, который обертывает фрагменты кода, в JShell разрешается также создавать свои

собственные классы. Более того, можно создавать объекты пользовательских классов, что позволяет экспериментировать с классами в интерактивной среде JShell. Этот процесс проиллюстрирован в следующем примере.

Начните новый сеанс JShell и введите код следующего класса, строка за строкой.

```
class MyClass {
   double v;

   MyClass(double d) { v = d; }

   // Возврат обратного значения v
   double reciprocal() { return 1.0 / v; }
}
```

По завершении ввода кода интерпретатор JShell выдаст следующее сообщение:

```
|   created class MyClass
```

Теперь, когда класс `MyClass` добавлен, можно использовать его. Например, можно создать объект `MyClass` с помощью следующей строки кода:

```
MyClass ob = new MyClass(10.0);
```

JShell сообщит, что была добавлена переменная `ob` типа `MyClass`. Затем введите следующую строку:

```
System.out.println(ob.reciprocal());
```

JShell выдаст значение `0.1`.

Интересно, что, когда вы добавляете класс в JShell, он становится статическим вложенным членом синтетического класса.

Использование интерфейса

Поддержка интерфейсов в JShell реализована таким же образом, как и поддержка классов. В частности, можно объявить интерфейс и реализовать его с помощью класса в JShell. Рассмотрим простой пример. Прежде чем начать, откройте новый сеанс JShell.

В данном интерфейсе объявляется метод `isLegalVal()`, который используется для определения корректности предоставленного значения. Метод возвращает `true`, если значение корректно, и `false` в противном случае. Допустимость значения будет определяться конкретным классом, реализующим интерфейс. Начнем с ввода следующего кода интерфейса в JShell.

```
interface MyIF {
   boolean isLegalVal(double v);
}
```

JShell ответит следующим образом:

```
|   created interface MyIf
```

Затем введем код следующего класса, который реализует интерфейс `MyIF`.

```
class MyClass implements MyIF {

   double start;
   double end;

   MyClass(double a, double b) { start = a; end = b; }

   // Определяет, находится ли v в диапазоне
   // от start до end включительно
   public boolean isLegalVal(double v) {
      if((v >= start) && (v <= end)) return true;
      return false;
   }

}
```

В ответ на это JShell выдаст следующее сообщение:

```
|   created class MyClass
```

Обратите внимание на то, что класс `MyClass` реализует метод `isLegalVal()`, проверяя, находится ли значение `v` в диапазоне значений переменных `start` и `end` экземпляра `MyClass`.

После добавления интерфейса `MyIF` и класса `MyClass` можно создать объект `MyClass` и вызвать для него метод `isLegalVal()`, как показано ниже.

```
MyClass ob = new MyClass(0.0, 10.0);

System.out.println(ob.isLegalVal(5.0));
```

В этом случае отображается `true`, поскольку значение 5 находится между 0 и 10.

Можно также создать ссылку на объект типа `MyIF`. Вот еще один пример корректного кода.

```
MyIF ob2 = new MyClass(1.0, 3.0);
boolean result = ob2.isLegalVal(1.1);
```

В этом случае переменная `result` принимает значение `true`, о чем сообщает JShell.

И еще одно: перечисления и аннотации поддерживаются в JShell точно так же, как классы и интерфейсы.

Оценка выражений и использование встроенных переменных

В JShell имеется возможность непосредственно вычислять выражения, которые не обязаны быть частью завершенных инструкций Java. Это особенно полезно в том случае, когда вы экспериментируете с кодом и хотите проверять выражения отдельно от программного контекста. Рассмотрим простой пример. Откройте новый сеанс JShell и введите следующую строку:

```
3.0 / 16.0
```

В ответ JShell выдаст следующее:

```
$1 ==> 0.1875
```

Как видите, был вычислен и отображен результат выражения. Также обратите внимание на то, что это значение было присвоено временной переменной `$1`. Всякий раз, когда выражение оценивается непосредственно, результат сохраняется во временной переменной соответствующего типа. Все имена временных переменных начинаются с символа `$`, за которым следует число, увеличивающееся с каждой новой переменной. Временные переменные можно использовать так же, как и любые другие переменные. Например, следующая строка кода отображает значение переменной `$1`, которое в данном случае равно `0.1875`:

```
System.out.println($1);
```

Вот еще один пример:

```
double v = $1 * 2;
```

Здесь в переменную `v` записывается результат умножения переменной `$1` на 2. В итоге переменная `v` будет содержать `0.375`.

Значение временной переменной можно изменить. Например, следующее выражение инвертирует знак переменной `$1`:

```
$1 = -$1
```

JShell отвечает так:

```
$1 ==> -0.1875
```

Выражения не ограничиваются числовыми значениями. Например, следующее выражение выполняет конкатенацию строки со значением, возвращаемым функцией `Math.abs($1)`:

```
"Абсолютное значение $1 равно " + Math.abs($1)
```

В результате создается еще одна временная переменная, содержащая строку:

```
Абсолютное значение $1 равно 0.1875
```

Импорт пакетов

Как отмечалось в главе 8, с помощью инструкции `import` делаются доступными классы, содержащиеся в пакете. Более того, всякий раз, когда используется пакет, отличающийся от `java.lang`, его нужно импортировать. В JShell похожая ситуация, за исключением того, что по умолчанию интерпретатор автоматически импортирует несколько часто используемых пакетов, среди которых — `java.io` и `java.util`. В результате вам не придется явно обращаться к инструкции `import`, чтобы подключать их.

Например, благодаря автоматическому импорту пакета `java.io` можно использовать следующую инструкцию:

```
FileInputStream fin = new FileInputStream("myfile.txt");
```

Вспомните, что класс FileInputStream содержится в пакете java.io. А поскольку пакет java.io автоматически импортирован, к его содержимому можно обращаться напрямую без явного включения инструкции import. Если в текущей папке содержится файл myfile.txt, JShell создаст переменную fin и откроет файл. Можно прочитать и отобразить содержимое этого файла, использовав следующий код.

```
int i;

do {
   i = fin.read();
   if(i != -1) System.out.print((char) i);
} while(i != -1);
```

Это тот же простейший код, который рассматривался в главе 10, только на этот раз не нужно явно указывать инструкцию import java.io.

Учитывайте, что JShell автоматически импортирует лишь некоторые часто используемые пакеты. Если требуется пакет, который не был автоматически импортирован интерпретатором, его придется импортировать явно, как в обычной программе Java. И еще одно: если нужно просмотреть список текущих операций импорта пакетов, воспользуйтесь командой /imports.

Исключения

В примере операций ввода/вывода, рассмотренном в предыдущем разделе, был проиллюстрирован другой очень важный аспект JShell. Обратите внимание на то, что здесь отсутствуют блоки try/catch, которые обрабатывают исключения, связанные с вводом/выводом. Если вы вернетесь к аналогичному примеру кода в главе 10, то увидите, что код, который открывает файл, перехватывает исключение FileNotFoundException, а код, который считывает содержимое файла, отслеживает исключение IOException. В данном случае отслеживать эти исключения не нужно, поскольку JShell автоматически выполняет всю работу за вас. Более того, зачастую JShell автоматически обрабатывает проверяемые исключения.

Другие команды JShell

Помимо ранее рассмотренных команд, JShell поддерживает и ряд других. Вот команда, которую вам стоит опробовать немедленно: /help. Она выводит список всех доступных команд. Для получения справки можно также ввести команду /?. Рассмотрим еще несколько часто используемых команд.

Для сброса JShell предназначена команда /reset. Она особенно полезна в том случае, когда нужно начать новый проект. Благодаря команде /reset не требуется выходить из интерпретатора JShell и запускать его заново. Но будьте осторожны, поскольку команда /reset выполняет сброс всей среды JShell, что приведет к потере информации о состоянии.

Для сохранения сеанса служит команда `/save`. В простейшем виде она выглядит так:

```
/save имя_файла
```

где *имя_файла* — это имя файла, в котором сохраняется информация о сеансе. По умолчанию команда `/save` сохраняет текущий исходный код, но это регулируется тремя опциями, две из которых представляют для нас интерес. С помощью опции `-all` можно сохранить все введенные строки кода, включая те, которые введены некорректно. С помощью опции `-history` можно сохранить историю сеанса (т.е. список введенных команд).

Для загрузки сохраненного сеанса предназначена команда `/open`, которая имеет следующий синтаксис:

```
/open имя_файла
```

где *имя_файла* — это имя загружаемого файла.

В JShell поддерживается ряд команд, которые позволяют выводить различные списки программных элементов. Эти команды приведены в следующей таблице.

Команда	Назначение
`/types`	Отображение классов, интерфейсов и перечислений
`/imports`	Отображение инструкций импорта
`/methods`	Отображение методов
`/vars`	Отображение переменных

Например, введите следующие строки кода.

```
int start = 0;
int end = 10;
int count = 5;
```

Если теперь ввести команду `/vars`, то вы получите такой результат.

```
|   int start = 0;
|   int end = 10;
|   int count = 5;
```

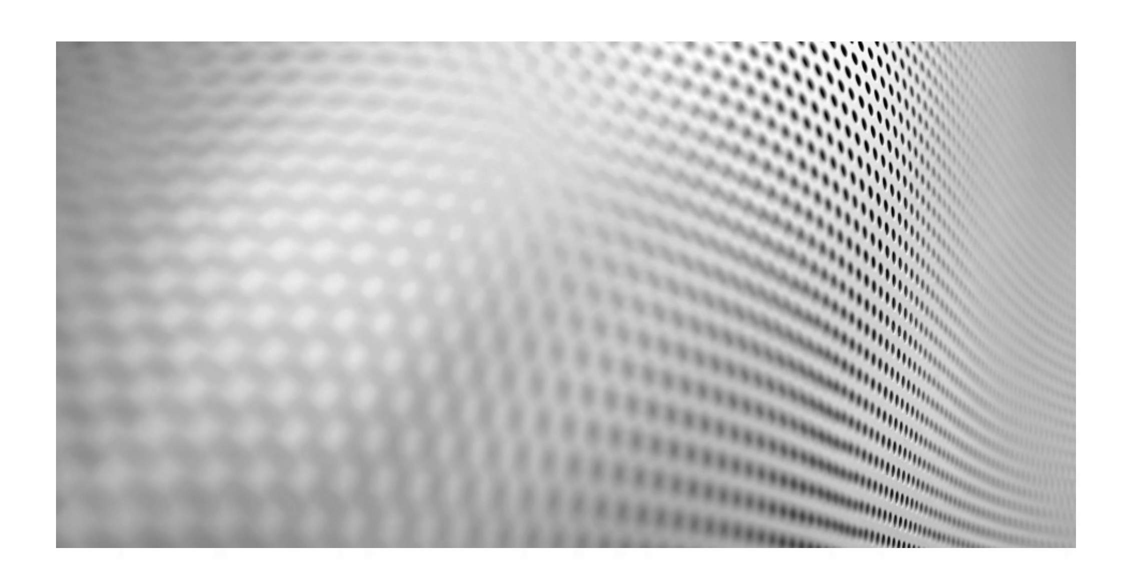
Еще одна полезная команда — `/history`. Она предназначена для просмотра истории текущего сеанса. История включает список всего, что было введено в командной строке.

Дальнейшее изучение JShell

Лучший способ изучить интерпретатор JShell — начать работать с ним. Попробуйте ввести несколько инструкций Java и понаблюдайте за реакцией JShell. Продолжайте экспериментировать, чтобы определить для себя, как эффективнее всего применять JShell в процессе разработки. Помните, что JShell

предназначен не только для новичков. Интерпретатор может применяться и для прототипирования кода. По мере изучения Java вы будете открывать для себя все новые возможности JShell.

Изучите также команды и параметры JShell. Поскольку JShell — новый инструмент, вполне может оказаться, что, когда вы будете читать эту книгу, у него появятся новые команды. Также не исключено, что средства JShell будут включены в интегрированные среды разработки Java, чтобы упростить процесс прототипирования/разработки. JShell является важным инструментом, который способствует дальнейшему совершенствованию навыков работы с Java.



Приложение Д

Дополнительные сведения о ключевых словах Java

Есть еще шесть ключевых слов Java, которые не были рассмотрены в книге:

- `transient`
- `volatile`
- `instanceof`
- `native`
- `strictfp`
- `assert`

Эти ключевые слова часто используются в более сложных программах, чем те, что описаны в книге. Тем не менее имеет смысл хотя бы кратко ознакомиться с ними, чтобы понимать их назначение. Кроме того, здесь будет описана другая форма ключевого слова `this`.

Модификаторы `transient` и `volatile`

Ключевые слова `transient` и `volatile` являются модификаторами типа, которые используются в особых случаях. Если переменная экземпляра объявлена как `transient`, то ее значение не должно сохраняться при сохранении объекта. Таким образом, поле `transient` не влияет на сохраняемое состояние объекта.

Модификатор `volatile` сообщает компилятору о том, что переменная может быть неожиданно изменена другими частями вашей программы. Подобное может происходить в многопоточных программах, когда два или более потоков используют одну и ту же переменную. Исходя из соображений эффективности каждый поток может сохранять свою собственную, частную копию такой общей переменной, возможно, в регистре ЦП. Основная (главная) копия переменной обновляется в разное время, например когда выполняется вход в метод `synchronized`. Такой подход вполне допустим, но бывают ситуации, когда он неуместен. Порой требуется, чтобы основная копия переменной всегда отражала текущее состояние, разделяемое всеми потоками. Чтобы гарантировать это, переменную нужно объявить как `volatile`.

Оператор `instanceof`

Иногда бывает полезно узнать тип объекта во время выполнения программы. Например, у вас может быть один поток выполнения, который генерирует различные типы объектов, и другой поток, который обрабатывает эти объекты. В такой ситуации полезно, чтобы поток обработки определял тип каждого объекта при его получении. Другая ситуация, когда важно знать тип объекта на этапе выполнения, связана с приведением типов. В Java некорректное приведение типов влечет за собой ошибку времени выполнения. Многие случаи, связанные с некорректным приведением типов, могут быть перехвачены на этапе

компиляции. Но когда создаются иерархии классов, некоторые ошибки приведения типов могут быть обнаружены только во время выполнения программы. Поскольку ссылка на суперкласс может указывать на объект подкласса, не всегда можно заранее знать, будет ли допустимым приведение типов, включающее ссылку на суперкласс. Для подобных ситуаций как раз и предназначен оператор `instanceof`, имеющий следующий синтаксис:

```
объектная_ссылка instanceof тип
```

Здесь *объектная_ссылка* — это ссылка на экземпляр класса, а *тип* — это тип класса или интерфейса. Если объект, адресуемый ссылкой, имеет указанный тип или может быть приведен к указанному типу, оператор `instanceof` вернет значение `true`. Иначе будет получен результат `false`. Как видите, оператор `instanceof` является средством для получения информации о типе объекта на этапе выполнения.

Ключевое слово strictfp

Ключевое слово `strictfp` является одним из наиболее экзотических. Когда много лет назад появилась версия Java 2, модель вычислений с плавающей точкой была слегка ослаблена. В частности, новая модель не требует усечения определенных промежуточных значений, создаваемых в процессе вычислений. В некоторых случаях это предотвращает переполнение или потерю точности. Включив в объявление класса, метода либо интерфейса ключевое слово `strictfp`, вы сможете гарантировать, что вычисления с плавающей точкой (и все связанные с ними усечения) будут выполняться с такой же точностью, как и в ранних версиях Java. Если класс объявлен с модификатором `strictfp`, последний распространяется на все методы в классе.

Инструкция assert

Инструкция `assert` используется для создания ***утверждения***, т.е. условия, которое, как ожидается, будет истинным во время выполнения программы. Например, у вас есть метод, который всегда должен возвращать положительное целое значение. Чтобы протестировать этот метод, достаточно создать утверждение с помощью инструкции `assert`, возвращающее значение больше нуля. Если на этапе выполнения программы условие действительно истинно, то никакие другие действия не выполняются. Если же условие ложно, генерируется исключение `AssertionError`. Исключения часто применяются в ходе тестирования для проверки соблюдения определенных условий. В производственном коде их обычно не используют.

Инструкция `assert` имеет два варианта синтаксиса. Первый вариант таков:

```
assert условие;
```

где *условие* — это выражение, результат оценки которого является булевым. Если выражение истинно, значит, утверждение верно и не нужно

предпринимать никаких действий. Если выражение ложно, значит, утверждение неверно, поэтому генерируется исключение AssertionError. Например:

```
assert n > 0;
```

Если n меньше или равно нулю, возбуждается исключение AssertionError. В противном случае никакие действия не выполняются.

Второй вариант инструкции assert показан ниже:

```
assert условие : выражение;
```

В данной версии *выражение* — это значение, которое передается конструктору AssertionError. Оно преобразуется в строковый формат и отображается в том случае, если утверждение неверно. Как правило, указывается строковое значение, но разрешается любое непустое выражение, если его можно преобразовать в строку.

Чтобы включить проверку утверждений на этапе выполнения, нужно задать параметр -ea, например:

```
java -ea Sample
```

Утверждения — полезное средство процесса разработки, поскольку они упрощают проверку ошибок в ходе тестирования. Но будьте осторожны: не следует полагаться на утверждение, чтобы выполнить какое-либо действие, действительно требуемое программой. Причина заключается в том, что обычно производственный код запускается с отключенными утверждениями, и выражения в утверждениях не будут вычисляться.

Собственные методы

Хоть и редко, но порой приходится вызывать подпрограммы, написанные не на Java. Как правило, такая подпрограмма будет предоставляться как исполняемый код для процессора и среды, в которой вы работаете, т.е. как платформо-зависимый (собственный) код. Например, можно вызвать подпрограмму собственного кода, чтобы ускорить выполнение, или использовать специализированную стороннюю библиотеку, скажем, статистический пакет. Это кажется невозможным, ведь Java-программы скомпилированы в байт-код, который затем интерпретируется (или компилируется на лету) исполняющей средой Java. К счастью, выход есть. В Java имеется ключевое слово native, которое используется для объявления собственных методов кода. Будучи объявленными, эти методы могут быть вызваны из Java-программы точно так же, как вызывается любой другой метод Java.

Чтобы объявить собственный метод, укажите для него модификатор native, но не задавайте тело метода. Например:

```
public native int meth() ;
```

Далее необходимо выполнить довольно сложную последовательность действий, чтобы связать реализацию собственного метода с вашим Java-кодом.

Другая форма ключевого слова this

Существует другая форма ключевого слова this, которая позволяет одному конструктору вызывать другой конструктор в том же классе. Общий синтаксис ключевого слова this, используемого в данном случае, выглядит так:

```
this(список_аргументов)
```

При вызове метода this() сначала выполняется перегруженный конструктор, который соответствует заданному *списку_аргументов*. Затем будут выполнены остальные инструкции, содержащиеся в исходном конструкторе. Вызов метода this() должен быть первой инструкцией в конструкторе. Ниже приведен простой пример.

```
class MyClass {
   int a;
   int b;

   // Раздельная инициализация переменных a и b
   MyClass(int i, int j) {
      a = i;
      b = j;
   }

   // Использование метода this() для инициализации
   // переменных a и b одним значением
   MyClass(int i) {
      this(i, i); // вызов конструктора MyClass(i, i)
   }
}
```

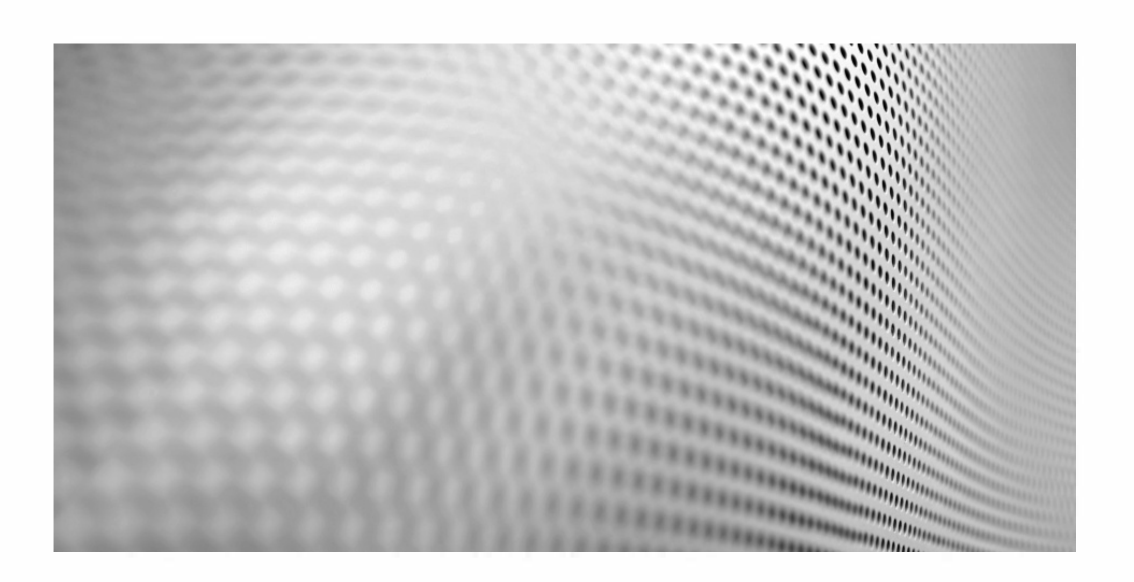
В классе MyClass только первый конструктор фактически присваивает значения переменным a и b, а второй конструктор просто вызывает первый. Таким образом, когда выполняется следующая инструкция:

```
MyClass mc = new MyClass(8);
```

вызов MyClass(8) приводит к выполнению вызова this(8, 8), который преобразуется в вызов MyClass(8, 8).

Вызов перегруженных конструкторов с помощью метода this() может быть полезным, поскольку предотвращает нежелательное дублирование кода. Но следует соблюдать осторожность. Конструкторы, вызывающие метод this(), могут выполняться немного медленнее, чем конструкторы, которые содержат весь встроенный код инициализации. Это связано с тем, что вызов второго конструктора влечет за собой дополнительные издержки. Помните о том, что создание объекта затрагивает всех пользователей вашего класса. Если класс применяется для создания большого количества объектов, может оказаться, что преимущество компактности кода будет нивелировано увеличением времени, которое тратится на создание объектов. По мере приобретения опыта программирования на Java вы научитесь принимать правильные решения в подобных ситуациях.

При использовании метода `this()` следует учитывать два типа ограничений. Во-первых, в вызове метода `this()` нельзя использовать никакие переменные экземпляра класса конструктора. Во-вторых, невозможно использовать методы `super()` и `this()` в одном и том же конструкторе, поскольку каждый из них должен быть первой инструкцией в конструкторе.



Приложение E

Знакомство с JDK 10

В этом приложении рассматриваются два важных средства, появившихся в Java с выходом JDK 10. Ранее между основными выпусками Java обычно проходило два года или даже больше. Но после выхода версии Java SE 9 (JDK 9) интервал времени между основными выпусками Java сократился. Начиная с версии Java SE 10 (JDK 10) выпуски версий будут появляться в соответствии с ускоренным графиком, причем интервал между основными выпусками будет составлять всего шесть месяцев.

Каждый основной выпуск, который теперь называется *функциональным выпуском*, будет включать функциональные средства, готовые на момент выпуска. Благодаря ускоренному графику выпусков программисты на Java смогут своевременно получать доступ к новым функциям и улучшениям. Кроме того, это позволяет авторам языка быстрее реагировать на постоянно меняющиеся требования сообщества Java. Проще говоря, ускорение графика выпусков сделано в интересах Java-программистов.

На момент написания книги функциональные выпуски запланированы на март и сентябрь каждого года. В результате комплект JDK 10 был выпущен в марте 2018 года, т.е. через шесть месяцев после выпуска JDK 9. Следующий выпуск запланирован на сентябрь 2018 года. Из-за более быстрого графика некоторые выпуски указываются в качестве *долгосрочной поддержки* (LTS). Это означает, что такой выпуск будет поддерживаться в течение определенного периода времени. Другие функциональные выпуски считаются краткосрочными. В частности, JDK 9 и JDK 10 обозначены как краткосрочные, и ожидается, что версия JDK 11, которая появится в сентябре 2018 года, станет LTS. Дополнительные сведения по этой теме можно найти в документации Java.

Среди всех новинок, связанных с появлением JDK 10, наибольший интерес для программистов на Java представляют две, которые и рассматриваются в этом приложении. Первая из них называется *выведением типа локальной переменной*. Это особенно важное средство, поскольку оно влияет как на синтаксис, так и на семантику языка Java. Вторая новинка, которая появилась в выпуске JDK 10, связана с изменениями в схеме нумерации версий JDK. Данные изменения отражают новый график версий и новую трактовку элементов номера версии. Это оказывает влияние на класс `Runtime.Version`, в котором инкапсулируется информация о версии.

Помимо двух основных новинок, описанных здесь, JDK 10 включает другие улучшения и изменения, в том числе несколько изменений в Java API. Подробно ознакомиться с этими изменениями можно в документации и в заметках к выпуску. Кроме того, следует тщательно изучать каждый выпуск, появляющийся раз в шесть месяцев. Как правило, все они включают ряд интересных обновлений.

Выведение типов локальных переменных

Начиная с JDK 10 тип локальной переменной может выводиться из типа ее инициализатора, а не указываться явно. Для поддержки этой новинки в Java был добавлен контекстно-зависимый идентификатор `var` в качестве зарезервированного имени типа. Выведение типов помогает упрощать код, устраняя необходимость избыточно указывать тип переменной там, где он может быть выведен из ее инициализатора. Это также позволяет упростить объявления в тех случаях, когда тип трудно распознать или когда он не может быть обозначен. (В качестве примера можно назвать тип анонимного класса.) Выведение типов локальных переменных стало неотъемлемым элементом современной практики программирования. Его включение в Java стало закономерной реакцией на передовые тенденции разработки языков программирования.

Чтобы использовать выведение типа локальной переменной, нужно объявить переменную с помощью ключевого слова `var`, используемого в качестве имени типа. Объявление переменной должно также включать инициализатор. Например, в старых версиях языка локальную переменную `counter` типа `int`, которая инициализируется значением 10, нужно было объявлять следующим образом:

```
int counter = 10;
```

При использовании выведения типа объявление можно переписать так:

```
var counter = 10;
```

В обоих случаях переменная `counter` имеет тип `int`. Только в первом случае тип указан явно, во втором он выводится как `int`, поскольку инициализатор 10 имеет тип `int`.

Как уже упоминалось, ключевое слово `var` добавлено как контекстно-зависимый идентификатор. Когда оно используется в качестве имени типа в контексте объявления локальной переменной, оно сообщает компилятору о том, что нужно использовать выведение типа для определения типа объявляемой переменной на основе типа инициализатора. Таким образом, в объявлении локальной переменной `var` является заполнителем для фактического, выводимого типа. Но в большинстве других контекстов `var` — это просто определяемый пользователем идентификатор, не имеющий специального назначения. Например, следующее объявление по-прежнему допустимо:

```
int var = 1; // здесь var - это просто идентификатор,
             // определяемый пользователем
```

В данном случае тип явно указывается как `int`, а `var` — это имя объявляемой переменной. Несмотря на то что это контекстно-зависимый идентификатор, в некоторых случаях он является недопустимым. В частности, он не может использоваться в качестве имени класса, интерфейса, перечисления или аннотации.

Следующая программа иллюстрирует вышесказанное.

```
// Простая демонстрация выведения типа локальной переменной
class VarDemo {
   public static void main(String args[]) {
      // Выведение типа используется для определения типа
      // переменной counter. В данном случае она имеет тип int.
      var counter = 10;
      System.out.println("Значение переменной counter: " + counter);

      // В следующем контексте var не является предопределенным
      // идентификатором. Это просто имя переменной, задаваемое
      // пользователем.
      int var = 1;
      System.out.println("Значение переменной var: " + var);

      // В следующем фрагменте обозначение var используется
      // и как указание на тип переменной, и как имя
      // переменной в инициализаторе
      var k = -var;
      System.out.println("Значение переменной k: " + k);
   }
}
```

Вот результат выполнения программы.

```
Значение переменной counter: 10
Значение переменной var: 1
Значение переменной k: -1
```

В предыдущем примере ключевое слово `var` применялось для объявления простых переменных, но его также можно использовать и для объявления массива. Например:

```
var myArray = new int [10]; // корректное объявление
```

Обратите внимание на то, что ни `var`, ни `myArray` не содержат квадратных скобок. В данном случае тип массива `myArray` выводится как `int[]`. Более того, в левой части объявления `var` *нельзя* использовать квадратные скобки. Следующие два объявления массивов являются недопустимыми.

```
var[] myArray = new int[10]; // недопустимо
var myArray[] = new int[10]; // недопустимо
```

В первой строке делается попытка указать квадратные скобки для объявления массива типа `var`, во второй строке — для объявления массива `myArray`. В обоих случаях использование квадратных скобок некорректно, поскольку тип выводится из типа инициализатора.

Важно подчеркнуть, что ключевое слово `var` может применяться для объявления переменной только в том случае, если переменная инициализируется. Например, следующая инструкция недопустима:

```
var counter; // недопустимо: требуется инициализатор!
```

Также помните о том, что ключевое слово var может применяться только для объявления локальных переменных, но не полей, параметров или возвращаемых типов.

Выведение типов локальных переменных со ссылочными типами

Можно выводить типы локальных переменных не только для примитивных типов, но и для ссылочных типов, таких как классы. Вот простой пример, в котором определяется переменная myStr типа String:

```
var myStr = "Это строка";
```

Поскольку в качестве инициализатора используется строка, заключенная в кавычки, выводится тип String.

Как уже упоминалось, одним из преимуществ выведения типов локальных переменных является упрощение кода, и в случае ссылочных типов это упрощение наиболее очевидно. Например, рассмотрим следующее объявление в традиционном стиле:

```
FileInputStream fin = new FileInputStream("test.txt");
```

С помощью ключевого слова var объявление можно переписать следующим образом:

```
var fin = new FileInputStream("test.txt");
```

где тип переменной fin выводится как FileInputStream, поскольку такой тип имеет соответствующий инициализатор. Как видите, не нужно явно повторять имя типа. В результате объявление переменной fin становится короче, чем традиционная запись. Таким образом, использование ключевого слова var позволило упростить объявление. Благодаря выведению типов программисты избавляются от необходимости вводить длинные названия типов.

Разумеется, выведение типов локальных переменных можно применять и при работе с пользовательскими классами, как показано в следующем примере.

```
// Выведение типов в случае пользовательского класса
class MyClass {
   private int i;

   MyClass(int k) { i = k;}

   int geti() { return i; }
   void seti(int k) { if(k >= 0) i = k; }
}

class VarDemo2 {
   public static void main(String args[]) {
      var mc = new MyClass(10); // обратите внимание на использование
                                // ключевого слова var
```

```
      System.out.println("Значение i в объекте mc равно " + mc.geti());
      mc.seti(19);
      System.out.println("Значение i в объекте mc теперь " + mc.geti());
   }
}
```

Здесь тип переменной mc определяется как MyClass, поскольку этот тип имеет инициализатор. Результат выполнения программы будет таким.

```
Значение i в объекте mc равно 10
Значение i в объекте mc теперь 19
```

Выведение типов локальных переменных и наследование

Выведение типов локальных переменных может вызывать путаницу в случае иерархий наследования. Как объяснялось в книге, ссылка на суперкласс может фактически быть ссылкой на объект производного класса, и это характерная особенность полиморфизма в Java. Однако важно помнить, что выведенный тип переменной основан на объявленном типе ее инициализатора. Следовательно, если инициализатор имеет тип суперкласса, то это и будет тип переменной. И не имеет значения, является ли фактический объект, на который ссылается инициализатор, экземпляром производного класса. Рассмотрим следующую программу.

```
// При использовании наследования выведенный тип является
// объявленным типом инициализатора и может отличаться от типа
// объекта, на который в реальности ссылается инициализатор.

class MyClass
   // ...
}

class FirstDerivedClass extends MyClass {
   int x;
   // ...
}

class SecondDerivedClass extends FirstDerivedClass {
   int y;
   // ...
}

class VarDemo3 {

   // Вернуть объект одного из типов MyClass
   static MyClass getObj(int which) {
      switch(which) {
         case 0: return new MyClass();
         case 1: return new FirstDerivedClass();
         default: return new SecondDerivedClass();
      }
   }
```

```
    public static void main(String args [] ) {
        // Несмотря на то что метод getObj() возвращает разные типы
        // объектов в иерархии наследования MyClass, его объявленный
        // тип возвращаемого значения - MyClass. В результате во всех
        // трех случаях выводится тип MyClass, хотя при этом получаются
        // разные производные типы объектов.

        // Здесь метод getObj() возвращает объект MyClass
        var mc = getObj(0);

        // В этом случае возвращается объект FirstDerivedClass
        var mc2 = getObj(1);

        // А в этом случае возвращается объект SecondDerivedClass
        var mc3 = getObj(2);

        // Поскольку типы переменных mc2 и mc3 выведены как MyClass
        // (это тип значения, возвращаемого методом getObj()), ни
        // объект mc2, ни объект mc3 не могут получить доступ к полям,
        // объявленным в подразумеваемых классах FirstDerivedClass
        // и SecondDerivedClass
//      mc2.x = 10; // некорректно, поскольку в классе MyClass нет поля x
//      mc3.y = 10; // некорректно, поскольку в классе MyClass нет поля y
}
```

В программе создана иерархия классов, на вершине которой находится класс `MyClass`. Класс `FirstDerivedClass` является подклассом `MyClass`, а класс `SecondDerivedClass` — подклассом `FirstDerivedClass`. Затем программа использует выведение типов для создания трех объектных переменных, `mc`, `mc2` и `mc3`, путем вызова метода `getObj()`. Метод `getObj()` объявлен как возвращающий значение типа `MyClass` (суперкласс), хотя в реальности он возвращает объект типа `MyClass`, `FirstDerivedClass` или `SecondDerivedClass`, в зависимости от типа передаваемого аргумента. Тем не менее выведенный тип всякий раз определяется только типом, указанным в объявлении метода `getObj()`, а не фактическим типом полученного объекта.

Выведение типов локальных переменных и обобщенные типы

Как объяснялось в главе 13, для обобщенных типов уже поддерживается одна разновидность выведения типов, реализуемая с помощью угловых скобок <>, которые еще называются *ромбовидным оператором*. Но выведение типов локальных переменных тоже можно использовать с обобщенными классами. Например, в случае класса

```
class MyClass<T>
    // ...
}
```

допустимо следующее объявление локальной переменной:

```
var mc = new MyClass<Integer>();
```

В этом случае тип переменной mc выводится как MyClass<Integer>. Также обратите внимание на то, что использование ключевого слова var приводит к более короткому объявлению, чем в других случаях. Как правило, названия обобщенных типов довольно длинные и порой сложные. Благодаря ключевому слову var подобные объявления существенно сокращаются.

И еще один момент. Ключевое слово var нельзя использовать в качестве имени параметра типа. Например, следующая конструкция недопустима:

```
class MyClass<var> { // недопустимо
```

Выведение типов локальных переменных в циклах for и блоках try

Выведение типов локальных переменных не ограничивается отдельными объявлениями, рассмотренными в предыдущих примерах. Оно может также встречаться в циклах for и в инструкциях try с ресурсами. Рассмотрим отдельно каждый случай.

Выведение типов может применяться при объявлении и инициализации переменной цикла в традиционном цикле for, а также при указании итерационной переменной в цикле типа for-each. Следующая программа демонстрирует обе возможности.

```
// Использование выведения типов в циклах for
class VarDemo4 {
   public static void main(String args [])

      // Выведение типа переменной цикла
      System.out.print("Значения x: ");
      for(var x = 2.5; x < 100.0; x = x * 2)
         System.out.print(x + " ");

      System.out.println();

      // Выведение типа итерационной переменной
      int[] nums = { 1, 2, 3, 4, 5, 6};
      System.out.print("Значения в массиве nums: ");
      for(var v : nums)
         System.out.print(v + " ");

      System.out.println();
   }
}
```

А вот результат выполнения программы.

```
Значения x: 2.5 5.0 10.0 20.0 40.0 80.0
Значения в массиве nums: 1 2 3 4 5 6
```

В данном примере тип переменной цикла `x` выводится как `double`, поскольку это тип инициализатора переменной. Для итерационной переменной `v` выводится тип `int`, поскольку это тип элементов массива `nums`.

В инструкции `try` с ресурсами тип ресурса может выводиться из инициализатора этого ресурса. Например, следующая инструкция корректна.

```
try (var fin= new FileInputStream("test.txt"))
   // ...
} catch(IOException exc) { // ... }
```

Здесь для переменной `fin` выводится тип `FileInputStream`, поскольку это тип инициализатора данной переменной.

Ограничения ключевого слова `var`

При использовании ключевого слова `var` имеют место несколько ограничений, дополняющих упомянутые выше ограничения. В частности, разрешается одновременно объявлять только одну переменную, переменная не может использовать `null` в качестве инициализатора, и объявленная переменная не может использоваться в выражении инициализатора. Кроме того, в качестве инициализаторов не могут использоваться ни лямбда-выражения, ни ссылки на методы. И хотя тип массива можно объявить с помощью ключевого слова `var`, не разрешается использовать `var` вместе с инициализатором массива. Например, следующая инструкция корректна:

```
var myArray = new int [10] ; // корректно
```

А эта инструкция недопустима:

```
var myArray = { 1, 2, 3 }; // недопустимо
```

Выведение типов локальных переменных не может применяться для объявления типа исключения, перехватываемого блоком `catch`.

Обновление схемы нумерации версий JDK и класс `Runtime.Version`

С появлением комплекта JDK 10 схема нумерации версий JDK была изменена, чтобы лучше соответствовать более плотному графику выпуска новых версий. Ранее для нумерации версий JDK применялся широко известный подход *основная_версия.дополнительная_версия*. Но теперь он признан устаревшим. В результате элементы, образующие номер версии, получили другое назначение. Начиная с JDK 10 первые четыре элемента обозначают *счетчики*, идущие в следующем порядке: счетчик функциональных выпусков (feature release), счетчик промежуточных версий (interim release), счетчик выпусков обновлений (update release) и счетчик выпусков исправлений (patch release). Номера разделены точками, при этом завершающие нули вместе с предшествующими точками удаляются. В нумерацию могут быть включены и дополнительные элементы, но предопределено назначение лишь первых четырех элементов.

Счетчик функциональных выпусков определяет главный номер версии. Этот счетчик обновляется после появления каждой функциональной версии (в данный момент — каждые шесть месяцев). Чтобы смягчить переход от предыдущей схемы нумерации версий, значение счетчика функциональных выпусков начинается с 10. Например, значением этого счетчика для JDK 10 будет 10.

Счетчик промежуточных версий указывает номер выпуска, появляющегося между функциональными выпусками. На момент написания книги значение счетчика промежуточных версий равнялось нулю, поскольку промежуточные версии пока что не включены в график выпусков (этот счетчик зарезервирован на будущее). Промежуточная версия не должна вносить каких-либо серьезных изменений в JDK. Счетчик выпусков обновлений указывает номер выпуска, который касается устранения проблем безопасности и, возможно, ряда других проблем. Счетчик выпусков исправлений указывает номер выпуска, в котором устраняются серьезные проблемы, требующие скорейшего реагирования. С появлением каждого нового функционального выпуска счетчики промежуточных версий, обновлений и исправлений сбрасываются до нуля.

Следует отметить, что только что описанный номер версии является необходимым компонентом *строки версии*, которая может также включать необязательные элементы. Например, она может содержать информацию о предварительной версии (pre-release). Необязательные элементы указываются в строке после номера версии.

Класс `Runtime.Version` был добавлен в Java API с появлением версии JDK 9. И хотя он не рассматривался в книге, читателям будет полезно получить о нем хотя бы базовую информацию. Назначение класса `Runtime.Version` заключается в инкапсуляции информации о версии, относящейся к среде выполнения Java. В JDK 10 класс `Runtime.Version` был обновлен и теперь включает следующие методы, которые поддерживают значения новых счетчиков функциональных выпусков, промежуточных версий, выпусков обновлений и выпусков исправлений.

```
int feature()
int interim()
int update()
int patch()
```

Каждый метод возвращает целочисленное значение счетчика. Вот пример небольшой программы, демонстрирующий использование счетчиков.

```
// Демонстрация использования счетчиков из класса Runtime.Version
class VerDemo {
   public static void main(String args[]) {
      Runtime.Version ver = Runtime.version();

      // Отображение отдельных счетчиков
      System.out.println("Счетчик функциональных выпусков: " +
                         ver.feature());
```

```
        System.out.println("Счетчик промежуточных версий: " +
                           ver.interim());
        System.out.println("Счетчик выпусков обновлений: " +
                           ver.update());
        System.out.println("Счетчик выпусков исправлений: " +
                           ver.patch());
    }
}
```

Класс `Runtime.Version` также включает ряд других методов. Три из них представляют особый интерес, поскольку возвращают дополнительные данные о версии (при наличии таковых). Это методы `pre()`, `build()` и `optional()`, которые возвращают информацию о предварительном выпуске, номере сборки и другие необязательные данные соответственно. Если ваша программа должна получать доступ к строке версии Java, вам придется детально ознакомиться с возможностями класса `Runtime.Version`.

В связи с изменением графика выпусков следующие методы класса `Runtime.Version` перешли в категорию нерекомендуемых: `major()`, `minor()` и `security()`. Раньше они возвращали соответственно основной номер версии, дополнительный номер версии и номер обновления безопасности. Эти значения были заменены номерами функциональных выпусков, промежуточных версий и выпусков обновлений, как описано выше.

Предметный указатель

Н

О

П